Lab 5 — MCP and Skills

AI-Assisted Programming for PhD Researchers

Two things extend an agent past the code in front of it: **skills**, which hand it reusable instructions (here, a set of academic-writing editors plus one that looks up real papers), and **MCP servers**, which hand it new tools (here, GitHub's hosted server). This lab is five short exercises: run a writing skill over your report's prose, write your own skill from scratch, build a *verified* reading list with the reference-lookup skill, wire up GitHub's MCP server and use it read-only, and split work across two agents at once using a git worktree. The throughline is the same one from Lab 4: the tools do more, but you still confirm every claim before you trust it.

Budget about **50 minutes**. Continue in the same repository, on the state you reached at the end of Lab 4.

Task 1 — Writing Skills

Goal: install the course's academic-writing skills and run one over the prose in your report, reviewing each proposed edit rather than accepting a rewrite.

Get the skills by cloning the course repository and copying both the writing pack and the reference-lookup skill into your project's skill directory:

```
git clone https://github.com/beyondsimulations/gAI-Assisted-Programming course-site
mkdir -p .claude/skills
cp -r course-site/skills/writing/* .claude/skills/
cp -r course-site/skills/reference-lookup .claude/skills/
```

> **i** Note
>
> OpenCode discovers Claude Code-compatible skills at `.claude/skills/<name>/SKILL.md` (project-level), no conversion needed. After the copy you should have eight writing skills: `academic-grammar`, `conciseness`, `hedging-language`, `literature-grounder`, `humanizer`, `acronym-check`, `term-consistency`, and `tense-consistency`, plus the `reference-lookup` skill you'll use in Task 3.

Restart OpenCode (or start a fresh session) so it picks up the new skills, then point the **conciseness** skill at the prose introduction of the report you generated in Lab 4:

> **Prompt**
>
> Apply the conciseness skill to the introduction section of `out/report.md`. Show me the proposed edits as a diff. Don't write the file yet.

Read the diff edit by edit. Conciseness cuts wordiness ("in order to" → "to", redundant qualifiers), but a cut that removes a *number*, a hedge, or a qualifier can change what the sentence claims. **Accept selectively:** keep the edits that only tighten, reject any that alter meaning.

> **Checkpoint**
>
> The eight writing skills (and the `reference-lookup` skill) exist under `.claude/skills/`, the agent produced a diff against `out/report.md`'s introduction, and you have gone through it and decided edit-by-edit which to accept.

> **Explain it**
>
> Pick **one edit you rejected** and say why: what did the "concise" version drop (a unit, a number, a hedge, a distinction), and how would the tightened sentence mislead a reader compared to the original?

Task 2 — Write Your Own Skill

Goal: write a skill of your own from scratch, run it, and confirm its shape matches the ones you just installed.

A skill is nothing more than a folder with a `SKILL.md` inside it. Create `.claude/skills/figure-caption-checker/SKILL.md` with the template below:

```
---
name: figure-caption-checker
description: Checks that figure captions in a document state what is shown,
how it was measured, and the sample size, with units, and without smuggling
interpretation into the caption. Use when reviewing figures in a report
or manuscript.
---

Review each figure caption in the document and check that it:

- states **what** is shown (the variable and the comparison), not just a label;
- states **how** it was measured or computed (method, instrument, or
statistic);
- reports the **sample size** (n) behind the figure;
- carries **units** on every quantity mentioned;
```

```
- contains **no interpretation** — the caption describes the figure; conclusions belong in the text.

Report each caption that fails one or more of these as a short list: the figure, which check failed, and the minimal fix. Show your findings as a list, not a rewrite of the document.
```

Restart OpenCode (or start a fresh session) so it picks up the new skill, then run it against your report and ask for a list of problems, not a rewrite:

> **Prompt**
>
> Apply the figure-caption-checker skill to `out/report.md`. List the problems you find; don't edit the file.

Now compare the skill you wrote to the eight you installed in Task 1: open one of their `SKILL.md` files alongside yours. Every one has the same shape: a `name`, a `description` that says *when* it applies, and a short instruction body that ends by asking for a list, not a rewrite.

> **Checkpoint**
>
> `.claude/skills/figure-caption-checker/SKILL.md` exists, a fresh session loaded it, and the agent used it to return a list of caption problems (or a clean pass) against `out/report.md` without editing the file.

> **Explain it**
>
> The agent loaded your skill for *this* task but not for unrelated ones. **Which line made that happen, and how?** Point to the part of the file the agent reads to decide whether a skill applies, and say what would go wrong if it were vague or missing.

Task 3 — A Verified Reading List

Goal: use the `reference-lookup` skill to assemble a short, *checked* reading list — and prove to yourself that a returned DOI is not the same as a correct DOI.

You installed the `reference-lookup` skill alongside the writing pack in Task 1. It resolves a paper title to a DOI and year through the free, keyless OpenAlex API, then cross-checks that DOI against Crossref: no server to wire up, no key to manage. Restart OpenCode (or start a fresh session) so the skill is loaded, then ask the agent to do the lookup and write the file:

Prompt

Using the reference-lookup skill, find the DOI, publication year, and a one-sentence statement of relevance for these three papers, then write `docs/references.md` as a short list:

1. Wilson et al., "Good enough practices in scientific computing";
2. Sandve et al., "Ten simple rules for reproducible computational research";
3. one more paper **you** pick from your own field on data quality or reproducibility.

For each entry give: full title, authors, year, DOI, and one sentence on why it matters for a research pipeline.

Now **the rule that makes this a reading list and not a hallucination**:

The rule

Open every DOI in your browser and confirm it resolves to the paper the agent claimed. Type `https://doi.org/<DOI>` for each entry. Check that the title, authors, and year on the landing page match what's in `docs/references.md`, and that the one-sentence relevance is actually supported by the abstract, not invented. A returned DOI is a *candidate, not proof*: OpenAlex can surface a plausible-looking DOI for the *wrong* paper, or one that resolves to nothing. That's exactly why the skill cross-checks it against Crossref before handing the final check back to you. You only have a reference once you've seen it resolve.

Fix any entry where the DOI, year, or title didn't check out, and re-verify.

Checkpoint

`docs/references.md` lists three papers, and you have personally opened all three DOIs in a browser and confirmed each resolves to the paper named in the file.

Explain it

Did every DOI resolve to the claimed paper? If one didn't, say exactly what went wrong: was it a DOI that 404'd, a DOI that resolved to a *different* paper (which one?), a wrong year on a right paper, or a relevance sentence the abstract doesn't support? Name the specific failure. If all three were perfect, say how you'd have caught it if one had been off.

Task 4 — Wire GitHub MCP

Goal: add GitHub's hosted MCP server to your project config, authenticate it, and use it **read-only** to explore a public repository from inside a session.

MCP servers are declared in `opencode.json` at the repository root, under an `mcp` key. Unlike a `local` server that OpenCode starts with a command, GitHub runs a **hosted** server you connect to over the network, so the entry is a `remote` one pointing at its URL. Open (or create) `opencode.json` and add:

```
{
  "mcp": {
    "github": {
      "type": "remote",
      "url": "https://api.githubcopilot.com/mcp/",
      "enabled": true
    }
  }
}
```

If `opencode.json` already has other keys (a `model`, a `permission` block), merge the `mcp` key in alongside them rather than overwriting the file.

Now authenticate. The simplest path is OAuth. OpenCode opens a browser and you approve access:

```
opencode mcp auth github
```

If you would rather use a token, create a **read-only fine-grained personal access token** on GitHub, export it (`export GITHUB_PAT=…`), and pass it as a Bearer header instead of OAuth:

```
"github": {
  "type": "remote",
  "url": "https://api.githubcopilot.com/mcp/",
  "enabled": true,
  "headers": { "Authorization": "Bearer {env:GITHUB_PAT}" },
  "oauth": false
}
```

> **i** In class
>
> GitHub's MCP endpoint, its auth flow, and the exact config keys are version-sensitive and change often. **Verify these against the current GitHub MCP docs, or use the values announced in class**, before you restart. Keep the token read-only: a read-scoped token means the agent *cannot change anything*, on any repo.

Restart OpenCode so it re-reads the config, and confirm the server registered:

```
opencode mcp list
```

Then start a session and put the server to work **read-only** on a public repo you **don't** need to own: the course repo itself, `beyondsimulations/gAI-Assisted-Programming`, is a fine target:

> **Prompt**
>
> Using the GitHub MCP server, on the public repo `beyondsimulations/gAI-Assisted-Programming`: list the currently open issues, then summarize the most recently merged pull request in two sentences.

> **Checkpoint**
>
> `opencode mcp list` shows `github`, and inside a session the agent used its GitHub tools to return real open issues and a real merged-PR summary for a repo you do not own, without cloning it.

> **Explain it**
>
> The token you gave the server is read-only. **What could this same server do if you had granted it write scope**, and which of those actions would you never want an agent to take without confirming first? Name one read action you just relied on, and one write action you deliberately withheld.

Task 5 — A Parallel Worktree

Goal: run two agents at once on the same repository without them colliding, by giving the second one its own working directory via a git worktree.

A worktree is a second checkout of the same repo on its own branch. Create one for a documentation branch:

```
git worktree add ../starter-docs improve-docs
```

Open a **second terminal**, move into that worktree, and start a separate OpenCode session there:

```
cd ../starter-docs && opencode
```

Give that second agent a self-contained documentation task:

> **Prompt**
>
> Add a one-line docstring to every public function in `src/pipeline/`. Don't change any behaviour: docstrings only.

While that runs, go back to your **first** terminal and session and do something small there in parallel, for example, add a `--version` flag to the pipeline CLI:

> **Prompt**
>
> Add a `--version` flag to the pipeline CLI that prints the package version and exits.

Both sessions are editing the same repository at the same time, but in different directories on different branches, so neither steps on the other. When the docstring agent is done, commit its work on `improve-docs`, then bring it back into your main branch and clean up the worktree:

```
git switch main
git merge improve-docs
git worktree remove ../starter-docs
```

> **Checkpoint**
>
> `improve-docs` is merged into `main`, `src/pipeline/` functions now have docstrings, the CLI has a `--version` flag, and `git worktree list` no longer shows `../starter-docs`.
>
> The reference tag `lab-05-done` captures the durable Lab 5 deliverables: the verified `docs/references.md` reading list from Task 3 and the capstone `sample.bib`. The tag's `docs/references.md` is a five-entry verified list, a superset of the three you assembled yourself. The writing-skills work (Tasks 1 and 2, running a skill and writing your own), the GitHub MCP exploration (Task 4), and this worktree/docstring/`--version` practice (Task 5) are hands-on and stay in your own working copy; they are not part of the tag. Use the tag to catch up on the references work if you fell behind:
>
> ```
> git fetch --tags && git checkout tags/lab-05-done -b catchup-05
> ```

> **Explain it**
>
> **Restate the mechanism in your own words:** what would have happened if both sessions had worked in the *same* directory instead of separate worktrees? Describe concretely how one agent's edits or branch switch would have collided with the other's, and why giving the second agent its own worktree prevents it.

If You Are Ahead

Push your own skills further. Pick one:

- **Write a second, field-specific skill.** Something you actually repeat: a reference-format checker, a units-consistency pass, a notation linter for your subfield. Give it a sharp `description` and run it against `out/report.md`.

- **Chain two writing skills in one pass.** Ask the agent to run, say, **conciseness** and then **hedging-language** over the same paragraph, and show the combined diff. Read whether the second skill undoes or compounds the first one's edits, and decide which changes survive.
- **Grab a discipline-specific editor.** The course ships optional field-specific skills (`math-notation`, `algorithm-check`, `model-formulation`, `english-variant`) in `skills/writing-extras/`. Copy the one that fits your field into `.claude/skills/` and run it on `out/report.md`.

> **Note**
>
> That's it for this lab. If anything is unclear, ask now — the next session builds on this state. If you fell behind, use the checkpoint tag from the last checkpoint box to catch up.

Bibliography