WORKSHOP

AI Code Assistants for HPC on Palmetto

Carl Ehrett • Spring 2026

Today’s learning objectives

- Understand where AI code assistants help (and where they do not) in HPC workflows
- Choose safe tools and workflows based on data sensitivity (what not to paste)
- Debug a failed Slurm job using a repeatable ”log -> hypothesis -> fix -> rerun” loop
- Use AI assistance for performance and efficiency improvement (vectorize, parallelize, benchmark your code)
- Learn what AI coding assistant resources are available to help with Palmetto usage

Agenda (3 hours)

Block 1

Intro + framing + running example

HPC-safe AI use (security + privacy + redaction)

Tool landscape: ChatGPT Edu, Codex CLI, VS Code Codex IDE

Block 2

Lab 1: local dev + VS Code AI with the MC mini-project

Lab 2: debug failed Palmetto jobs with Codex CLI

Block 3

Lab 3: resource efficiency + benchmarking

Prompting patterns (”cookbook”) for HPC workflows

Wrap-up + pitfalls + Q and A

Setup

To participate fully today, you will need to have already done the following:

1. Be able to access to the Palmetto cluster via `ssh` in the terminal. (https://docs.rcd.clemson.edu/palmetto/connect/ssh/)
2. Complete the Palmetto self-guided Onboarding training (https://docs.rcd.clemson.edu/palmetto/onboarding/).
3. Accept your invitation to join the ChatGPT EDU workspace.
4. Install `codex` on your laptop. (https://developers.openai.com/codex/cli/)

Together, we will do the following additional setup.

1. In Palmetto, load and log into `codex`.
2. Get an API key so you can use our Palmetto-inferenced LLM for code assistance.
3. Alter your `~/.codex/config.toml` file settings.
4. Copy the workshop files to your home directory.

Running example: MC mini-project (estimating pi)

- Small Python project (`mc_sim/` + `scripts/run_sim.py`)
- Designed to be:
- Refactored (Lab 1)
- Used in batch job submission (Lab 2)
- Benchmarked (Lab 3)

```python
import random


def estimate_pi(n_samples: int) -> float:
    if n_samples <= 0:
        raise ValueError("n_samples must be a positive integer")

    inside = 0
    for _ in range(n_samples):
        x = random.random()
        y = random.random()
        if (x * x + y * y) <= 1.0:
            inside += 1

    return inside / n_samples

if __name__ == "__main__":
    print(f"π estimate (demo): {estimate_pi(10_000):.6f} (n=10000)")
```

simulate.py

Ground rule: assistants are powerful but fallible

- Treat output as a **hypothesis generator**, not an authority
- Expect the assistant to make **confident mistakes** (especially about Palmetto cluster specifics)
- **You** are responsible for:
  - Palmetto policy compliance
  - Correctness and reproducibility
  - Not leaking sensitive info

Mental model: a very fast junior collaborator with incomplete context.

Useful for drafts, hypotheses, and small diffs. Not a source of truth.

HPC-safe AI use: what's safe to share?

OK

toy examples

generic algorithms

sanitized logs

non-sensitive configs

Maybe

small snippets of real code

only if policy allows and no sensitive context

Not OK

PHI or IRB data

sponsor or proprietary code

secrets, keys, identifiable paths or data dumps

Data protections vary by tool (know before you paste)

Always ask 3 questions:

Practical rule **match the tool to the data classification**

- **Public / classroom code** -> most tools are fine
- **Research code / internal info** -> prefer org-managed tools with clear terms
- **Sensitive / regulated / proprietary** -> prefer Palmetto-local inference or do not paste

Data protections by tool (cheat sheet)

Example tool	Typical data path	Typical training posture*	Safe default
ChatGPT EDU (org-managed)	sent to vendor servers	**not used for model training** under EDU/enterprise terms	treat as external; redact secrets
Palmetto-local inference	stays on Palmetto	**no external training** (data stays internal)	preferred for sensitive logs/code
Cursor / GitHub Copilot / Claude Code	vendor cloud integrations	defaults may allow **training on your data**; opt-out is sometimes possible	assume it is training unless policy says otherwise

Tool landscape (high-level)

Tool	Best for	Key limitation
ChatGPT EDU (web)	explanations, drafts, one-off questions	does not know your cluster or repo unless pasted
Codex CLI (on Palmetto)	log-driven debugging, patch suggestions	can hallucinate cluster details
VS Code Codex IDE (local or OOD)	in-context code work, autocomplete code assistance	can hallucinate cluster details, can be difficult to set up

Quick decision rule: which tool should I use?

- ”I need to understand / design / ask a simple question” -> ChatGPT EDU
- ”I have a failed job + log” or ”I want explanations of code” -> Codex CLI (or any CLI assistant workflow)
- ”I am editing code and want inline help” -> VS Code Codex IDE extension

Setup assumptions (today)

- You can access Palmetto (via SSH)
- You can open/edit the MC project:
  - in Palmetto
  - locally on your laptop
- You can run a Slurm job and read its output log

Lab 1 (overview)

Local dev + VS Code Codex IDE (0:40-1:05)

- Goal: use an assistant to improve code quality **without losing correctness**
- We will:
  - diagnose a bug in our code causing unexpected poor performance
  - add docstrings/comments
  - refactor for clarity
  - vectorize
  - add **multicore** option
  - add warning for low sample size

[DEMO] Demo 1: Bug diagnosis, docstrings, and light refactor

1. Open `simulate.py` and run the baseline (see surprising output)
2. Why is the estimate wrong?
3. Add docstrings + clarify variable naming (no behavior changes)
4. Refactor into smaller functions (keep CLI stable)
5. Make a vectorized variant

[LAB] Lab 1 tasks

- **Task A:** Make the script optionally multicore (requires changes to both `.py` scripts), with a `--num_workers` optional argument that defaults to 1.
- **Task B:** Add a warning if `n_samples` is too low for a good estimate.
- Success criterion: your run uses multiple cores and produces a reasonable pi estimate when `n_samples` is high and a warning when `n_samples` is low - and you can explain the changes.

Lab 1 debrief (what you should notice)

- Assistants are often great at:
  - explanations and docstrings
  - refactor suggestions
  - planning and offering ideas for accomplishing goals
- But common failure modes include:
  - subtle code behavior changes
  - looks right but is slower or wrong
  - ignoring constraints you did not state explicitly (for example, keep vectorization as optional)
  - **Complicating your codebase, distancing you from an understanding of your code!**

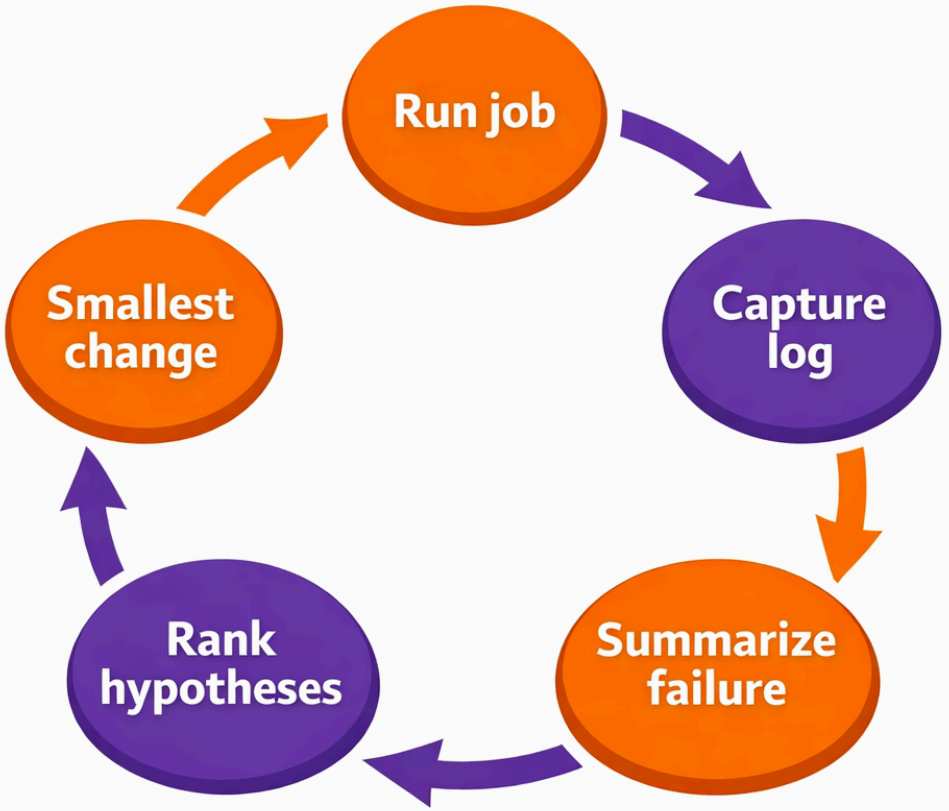
Lab 2 (overview)

Debugging failed Slurm jobs with Codex CLI

- Goal: practice using code assistants to diagnose Slurm job submission failures.
- We will:
  - submit a failing Slurm job
  - locate and read the job output or error
  - use Codex CLI to generate hypotheses + fixes
  - apply a fix and rerun

The Lab 2 debugging loop (the pattern)

1. Run job -> capture log
2. Summarize failure in plain language
3. List likely causes (ranked)
4. Propose the smallest change to test next
5. Rerun -> compare new log -> iterate



[DEMO] Demo 2: Slurm job failure debugging

1. `sbatch` a script which fails unexpectedly
2. Read the log to see why the job failed
3. Run Codex CLI with a structured prompt
4. Apply one targeted fix
5. Rerun and interpret the result

[LAB] Lab 2 tasks

Edit the slurm submission script and, if necessary, the `.py` files.

1. Make the script use number of workers equal to the number of cores available
2. Verify any env variables and other potentially incorrect aspects of the script
3. Verify using Jobstats that your job used all cores
4. Success criterion: you can explain how the change works, what changed, and what improved

Lab 2 debrief

- Where assistants hallucinate:
  - cluster-specific names (partitions, modules, paths)
  - policies ("you should do X")
- How to steer:
  - provide cluster facts explicitly
  - ask for uncertainty + alternatives
  - request verification steps ("what command checks this?")
  - give policy guidance ("be sure not to request a GPU")

Lab 3 (overview)

Resource efficiency + benchmarking (1:45-2:10)

- Goal: For quick, supplemental boilerplate code, save time by having a code assistant write it for you. Here, we write code to evaluate performance and measure it and produce human-digestible outputs.
- You will:
  - compare non-vectorized vs vectorized
  - compare single-core vs multi-core
  - produce a simple timing table and a plot

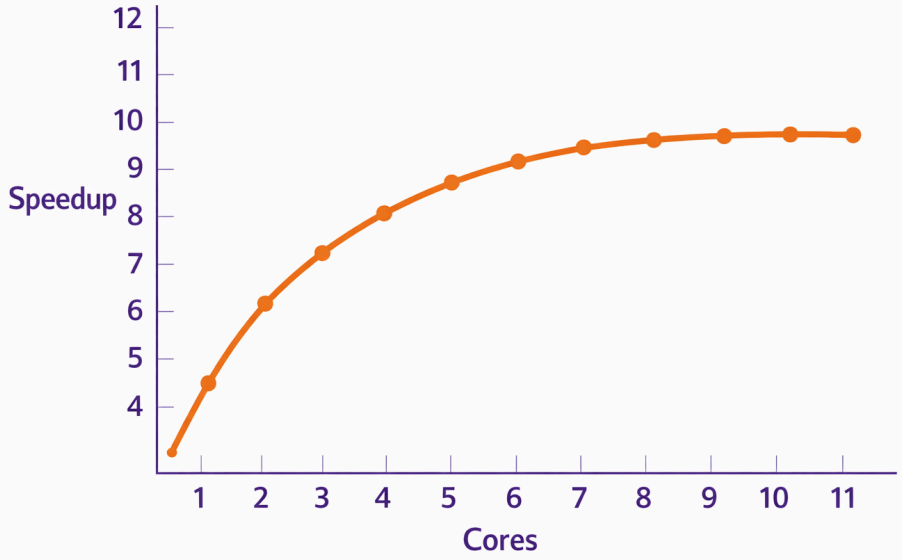
Lab 3: what you will measure

Matrix:

- vectorized vs non-vectorized
- single-core vs multi-core

Outputs:

- timing table (mean/std over repeated runs)
- sweep workers `1..N` and plot speedup



Runtime differences across setting configurations

- Examining directly some runs of the code using different settings, we can see outputs that suggest what we might expect in terms of speedup.
- But a systematic approach is more informative.

[LAB] Lab 3 tasks

Ask Codex to draft a new `.py` and a new `.sbatch` file, to do the following.

- Run all four configurations: vectorized and not, multicore and not, with 12 cores and 1M samples.
  - Run each configuration ten times.
  - Record the mean and standard deviation of runtime for each configuration.
  - Output a text file that contains a markdown table showing the table of means and sds.
- Run the vectorized, multicore configuration one time at each of all possible numbers of cores 1, 2, ..., 12, with `n_samples` set to 5M. Record the runtime for each run.
  - Produce a plot that displays the runtime for each choice of number of cores.
- Success criterion: you can understand the code that the assistant produces, and that code succeeds in creating the desired table and plot.

Best prompting practices (HPC code assistants)

- Start with the **artifact + objective**
  - ”Here is the `sbatch`, the command, and the error log; expected outcome is ___”
- Provide **just enough relevant context** to prevent guessing
  - modules or env, filesystem layout, Slurm constraints, how you run or retrieve logs
- State **constraints + guardrails**
  - keep CLI stable, minimal diff, do not touch unrelated files
- Ask for **structured output**
  - summary -> ranked hypotheses or options -> smallest next change (diff) -> how to validate
- Iterate deliberately
  - one change at a time; rollback plan; benchmark on the target system; **use version control**

Putting it all together: a recommended workflow

1. Draft locally with VS Code Codex IDE (small diffs)
2. Run locally (fast feedback)
3. Port to Palmetto + run jobs
4. Debug with log-first prompts (Codex CLI)
5. Optimize + benchmark
6. Document what worked (for reproducibility)

Pitfalls to avoid (and how to mitigate)

Pitfall	Mitigation
Pasting sensitive data	redact + policy check
Making huge changes blindly	demand diffs + small steps
Believing cluster-specific guesses	verify with commands or docs
Matching the resource request to what is needed	validate code; monitor small-scale jobs then large-scale jobs
Letting the assistant write large parts of the codebase	select carefully delimited boilerplate tasks to relegate to the assistant

Wrap-up + Q and A

- Questions or comments?