MAE 6291 – Class 7

Lorena A. Barba, March 17, 2026

AI-Assisted Coding: From Autocomplete to Agentic Engineering

In the first half of this course, we explored how AI can accelerate literature review, manage research knowledge, and extend what LLMs can do through tool use and context engineering. We also laid a foundation of technical understanding about how LLMs work, the emergence of reasoning models, and the agentic AI landscape. We now shift focus to a domain where AI is producing some of its most dramatic productivity gains: **writing code**.

This is especially relevant for you as engineering graduate students. Your research almost certainly involves computation—whether that means numerical simulations, data analysis pipelines, visualization scripts, or experimental control software. You are not computer scientists; code is a *tool* in service of your research, not an end in itself. That distinction matters, because the new generation of AI coding assistants is hugely impactful for this kind of user: someone who knows what they need to compute but would rather spend their time on the science than on painstakingly writing and debugging code.

To understand where we are now, we need to see how we got here. The timeline is short—barely five years—but the pace of change has been extraordinary.

A Brief History of AI-Assisted Coding

June 2021: GitHub Copilot – The Starting Gun

When GitHub (owned by Microsoft) released Copilot in June 2021, it was the first widely available AI coding assistant. Built on OpenAI's *Codex* model (a fine-tuned version of GPT-3), Copilot worked as an extension inside Visual Studio Code. As you typed, it would suggest completions—not just single lines, but entire functions—based on the context of your file and your comments.

The experience felt like magic, and perhaps a little unsettling. You could write a comment like `# calculate the Reynolds number for pipe flow` and Copilot would generate the function and show it as grey code right below; you could hit Tab to accept the code suggestion and carry on. Copilot was trained on billions of lines of public code from GitHub repositories, which meant

it had seen nearly every common programming pattern. For boilerplate code—reading CSV files, setting up Matplotlib plots, implementing standard algorithms—it was remarkably effective.

But Copilot in 2021 had clear limitations. It operated on a single file at a time, with no awareness of your broader project. It couldn't *run* code, check if its suggestions actually worked, or learn from errors. It was, in essence, a very good autocomplete: impressive for speeding up typing, but still firmly in the role of a simple assistant.

2023–2024: The IDE Redefined by AI – Cursor and Friends

By 2023, a new generation of AI tools appeared that took a fundamentally different approach. Instead of adding AI as a plugin to an existing editor, projects like **Cursor** rebuilt the editor itself around AI. Cursor is a fork of VS Code—it looks and feels familiar, and you can even import your VS Code extensions and keybindings—but its architecture was redesigned so the AI can "see" your entire codebase, not just the file you have open.

This was a meaningful leap. Cursor introduced what it called *Composer* mode, where you could describe a high-level task (like "add user authentication to this web app") and the AI would plan changes across multiple files, create new files, and edit existing ones simultaneously. The AI gained awareness of your project's structure, dependencies, and patterns.

Around the same time, other tools followed similar philosophies. **Windsurf** (by Codeium), **Cline** (an open-source VS Code extension), and later **Lovable** (focused on building web applications from natural language descriptions) all pushed the boundary from "autocomplete" toward something more like "AI pair programmer." The user increasingly described what they wanted rather than how to implement it.

February 2025: Claude Code – A Phase Change

In February 2025, Anthropic released **Claude Code** as a research preview (it became generally available in May 2025). Claude Code represented something qualitatively different from what came before. Rather than living inside an IDE, Claude Code runs in your *terminal*, the command line interface. It connects to a Claude model hosted on Anthropic's servers, and from there it can read your files, write new ones, execute shell commands, run tests, commit to Git, and iterate on errors. Rather than just suggesting code, it directly acts on your codebase.

This made Claude Code one of the first convincing demonstrations of an **agentic coding tool**: an AI system that doesn't wait passively for you to accept or reject suggestions, but instead plans multi-step strategies and executes them. Ask Claude Code to "refactor this module to use async/await and update all the tests," and it will read the current code, identify what needs to change, make the edits across multiple files, run the test suite, diagnose failures, and fix them—potentially through several rounds of iteration—before presenting you with the result.

The growth was explosive. By November 2025 (six months after its general release), Claude Code had surpassed $1 billion in annualized revenue.[1] By February 2026, that figure was reported as $2.5 billion, with 4% of all public GitHub commits estimated to be authored by Claude Code.[2] Professional developers everywhere are now delegating substantial portions of their workflow to it.

It was at the same moment that Claude Code was released in February 2025 when a cultural shift crystallized. Andrej Karpathy, the renowned AI researcher, former Director of AI at Tesla, and co-founder of OpenAI, posted what he later called a "shower of thoughts throwaway tweet" that went viral (over 6.5 million views). In it, he described what he called **"vibe coding"**:

> *"There's a new kind of coding I call 'vibe coding,' where you fully give in to the vibes, embrace exponentials, and forget that the code even exists. […] I just see stuff, say stuff, run stuff, and copy paste stuff, and it mostly works."*
>
> — Andrej Karpathy, February 2, 2025 on X (ex Twitter), https://x.com/karpathy/status/1886192184808149383

Karpathy described using Cursor's Composer with Anthropic's Sonnet model, even dictating instructions by voice, barely touching the keyboard. He accepted all changes without reading the diffs. When error messages appeared, he pasted them back to the AI without comment. When bugs proved stubborn, he asked for "random changes until it goes away."

The new term captured a real phenomenon; it entered popular usage so fast that Merriam-Webster Dictionary added it as a trending word. But it also provoked some backlash. Andrew Ng (ex Stanford professor, entrepreneur and investor) and many practicing engineers pushed back on the name, arguing that professional development requires more oversight and discipline. One year later, Karpathy himself reflected that while vibe coding was "fun for throwaway projects," the real transformation was "**agentic engineering**": programming via LLM agents as a default professional workflow, "with more oversight and scrutiny." He called it an art and science with its own depth, something you can *learn and become better at*.[3]

2026: The Current Landscape

We are now in a period of rapid maturation. Claude Code currently uses **Opus 4.6**, a model that leads on coding benchmarks (including Terminal-Bench 2.0 for agentic coding[4]) and offers a 1M token context window in beta, enough to hold an entire medium-sized codebase in memory. It exists within a growing ecosystem: **MCP** (Model Context Protocol; more on this later) lets Claude Code connect to external data sources and tools through standardized interfaces; the

[1] https://www.anthropic.com/news/anthropic-acquires-bun-as-claude-code-reaches-usd1b-milestone
[2] https://www.anthropic.com/news/anthropic-raises-30-billion-series-g-funding-380-billion-post-money-valuation
[3] https://x.com/karpathy/status/2019137879310836075
[4] Terminal-Bench 2.0 is an advanced, 89-task benchmark designed to evaluate agentic AI coding capabilities, on complex, real-world coding tasks that require multi-step reasoning, tool use, and autonomous problem-solving.

skills system (`SKILL.md` files; more on this later) lets you teach Claude Code domain-specific workflows; and **agent teams** allow multiple Claude instances to work on sub-tasks in parallel.

Meanwhile, every major AI lab now has a terminal-based coding agent. OpenAI released **Codex CLI**, a Rust-based open-source agent that runs locally and uses OpenAI's specialized Codex models (now on GPT-5.3-Codex). It's included with ChatGPT Plus subscriptions. Google released **Gemini CLI**, which stands out for its generous free tier (1,000 requests per day with just a Google account) and its massive context window. And GitHub itself has entered the arena: **GitHub Copilot CLI**, which reached general availability in February 2026,[5] transforms GitHub's original autocomplete assistant into a full terminal agent with Plan mode, Autopilot mode, MCP support, and skills—included with all Copilot subscriptions, including the Student Developer Pack you already have. These are not experimental toys, they represent the three largest AI companies converging on the same core idea: an AI agent that lives in your terminal, understands your code, and can act on it.

The IDE-based tools have also matured dramatically. Cursor, now at version 2.5, introduced *Automations* (always-on agents triggered by events like code commits or Slack messages), *Mission Control* (a dashboard for managing multiple agents), and its own fast coding model called *Composer*. Its revenue reportedly doubled to over $2 billion in three months. GitHub Copilot, the original, has expanded to include Claude Opus 4.6 as a selectable model and now offers agentic capabilities far beyond its 2021 autocomplete. Google has also entered the IDE space with **Antigravity**,[6] an agent-first development platform (also built on a VS Code fork) powered by Gemini 3 that introduces a *"Manager Surface"* for orchestrating multiple agents working asynchronously across different workspaces.[7] Along with AWS's **Kiro**, it signals that every major cloud provider now considers agentic coding a core product category.

The AI Assistant Landscape at a Glance

For an engineering researcher, the choice of coding assistant depends on where you work and what you're building. The table below provides a rough map of the current landscape, organized by the amount of power each tool gives you versus the learning curve required to use it.

Tool	Type	Power	Learning Curve	Best For
GitHub Copilot	IDE extension	Moderate	Low	Autocomplete, inline help, code explanation
Google Colab AI	Notebook feature	Low–Moderate	Very low	Quick code generation in notebooks, cell-level help

[5] https://github.blog/changelog/2026-02-25-github-copilot-cli-is-now-generally-available/
[6] https://antigravity.google/
[7] https://developers.googleblog.com/build-with-google-antigravity-our-new-agentic-development-platform/

Cursor	AI-native IDE	High	Moderate	Multi-file projects, web apps, agentic workflows
Lovable	Web builder	High (narrow)	Very low	Full-stack web apps from natural language; non-coders
Claude Code	Terminal agent	Very high	Moderate–High	Complex refactors, research automation, multi-step tasks
Codex CLI	Terminal agent	Very high	Moderate–High	OpenAI ecosystem users, lightweight local agent
Gemini CLI	Terminal agent	High	Moderate	Free tier, Google ecosystem, large-context tasks
GitHub Copilot CLI	Terminal agent	High	Moderate	GitHub-integrated workflows; free for students with Developer Pack

Three Tiers of AI Coding Assistance

It helps to think about these tools as falling into three broad tiers, each representing a different relationship between you and the AI.

Tier 1: Autocomplete and Inline Assistance

This is where it all started, and it remains the gentlest entry point. **GitHub Copilot** in VS Code and **Google Colab**'s built-in AI features both fall here. The AI watches what you type and suggests completions. You stay in full control: you see every suggestion, you accept or reject each one, and the AI never modifies your files without your explicit action.

For you, this tier is immediately accessible. If you've used VS Code even casually, enabling Copilot takes about two minutes (and it's free through the Student Developer Pack[8]). If you work in Jupyter notebooks via Google Colab, the AI features are already built into the interface. The learning curve is essentially zero.

The limitation is equally clear: these tools are reactive. They help you type faster, but they don't think about your project holistically. Copilot won't tell you that your approach to solving a PDE is inefficient, or that there's a better library for your use case. It generates code one suggestion at a time, and you assemble the larger picture.

Tier 2: AI-Native Editors

Cursor is the defining example. Unlike a plugin that adds AI to VS Code, Cursor *is* the editor: a full fork of VS Code rebuilt so the AI understands your entire repository. It can plan multi-file

[8] https://education.github.com/pack

changes, run tests, and iterate on errors. Its *Agent mode* lets it operate semi-autonomously: you describe a goal, and Cursor creates a plan, executes it, and asks for review.

The power increase is substantial. Where Copilot helps with individual lines or functions, Cursor can scaffold an entire feature across your project. It also integrates MCP servers, letting the AI connect to external tools and databases. As of early 2026, Cursor supports models from OpenAI, Anthropic, Google, and xAI, letting you choose the best model for each task.

The barrier is that Cursor replaces your editor. If you're comfortable in VS Code, the transition is smooth (your extensions and keybindings carry over), but you're still adopting a new application, learning new concepts like *Composer mode* and agent workflows, and managing a subscription ($20/month for Pro). For students already navigating multiple software tools, this is a non-trivial commitment.

Lovable represents a different kind of Tier 2 tool, optimized for building web applications through conversation rather than code. You describe what you want ("build me a dashboard that shows my experimental results with interactive plots") and Lovable generates a complete, deployable web app. It's extraordinarily powerful for its niche, but its scope is narrow: if you need to write a finite element solver or a data processing pipeline, Lovable is not the right tool.

Tier 3: Terminal Agents (Agentic Coding)

This is the frontier, and it's where the most dramatic capability gains are happening. **Claude Code, Codex CLI**, and **Gemini CLI** all operate in your terminal. They read your codebase, execute commands, modify files, run tests, and iterate, all through natural language conversation.

The power here is qualitatively different. A terminal agent does more than write code; it *reasons about your project and takes action*. You can say "Find all the places in this codebase where we compute pressure drop and make sure they're using consistent units," and the agent will search your files, identify the relevant functions, check the implementations, and propose (or directly make) corrections. You can say "Set up a GitHub Actions workflow that runs my test suite on every push," and it will create the configuration file, test it, and commit it.

Claude Code is currently the most capable of these tools, powered by Opus 4.6. It has the highest scores on agentic coding benchmarks, the deepest reasoning capabilities, and the most developed ecosystem (MCP, skills, agent teams). Codex CLI is notable for being open-source, built in Rust, and included with ChatGPT subscriptions. Gemini CLI offers the most generous free tier (1,000 requests per day) and Google's massive 1M+ token context window.

The learning curve is real, however. You need comfort with the terminal (or a willingness to learn). You need to understand enough about your project's structure to guide the agent effectively and review its output critically. You need to manage API costs (Claude Code charges per token; Codex CLI is included with ChatGPT Plus; Gemini CLI has a free tier). And you need to

develop judgment about when to trust the agent's output and when to intervene—the kind of judgment Karpathy called "agentic engineering."

What Changed: From Suggestion to Agency

The progression from Copilot (2021) to Claude Code (2025–2026) is much more than an improvement in capability—it's a real change in the relationship between the programmer and the tool. Understanding this shift is essential for using these tools effectively.

- With **autocomplete** (Copilot, Colab AI), you write code and the AI helps you type faster. The AI is like a scribe.
- With **AI-native editors** (Cursor), you describe features and the AI implements them across your project. The AI is more like a junior developer.
- With **terminal agents** (Claude Code, Codex CLI, Gemini CLI), you define goals and the AI plans, executes, tests, and iterates. The AI is now a capable colleague who happens to work very fast.

Each tier gives you more leverage but demands *more judgment*. With autocomplete, the worst that happens is a bad suggestion you ignore. With a terminal agent, the AI might restructure files, install dependencies, or push code to a repository. Your role shifts from writing code to *reviewing changes, setting guardrails, and making architectural decisions*. This is why all three terminal agents offer granular permission controls: you can choose how much autonomy the agent gets, from read-only consultation to full file and command access.

Implications for Engineering Researchers

As researchers in science and engineering, you occupy an interesting position. You're not professional software developers, but you may write a lot of code. Your code serves your research: it runs simulations, processes data, generates figures, and sometimes grows into tools that other researchers use. The quality, correctness, and reproducibility of that code matters greatly.

AI coding tools offer three concrete advantages for your situation:

Rapid prototyping of research tools. Need a script that parses your lab's data format, computes statistics, and generates publication-quality plots? A terminal agent can produce a working first draft in minutes that would take hours to write from scratch. Need a web dashboard to share results with collaborators? Lovable or Cursor (or even Gemini in Canvas mode for simple cases) can build one in an afternoon.

Overcoming unfamiliar territory. Your research might put you in contact with technologies outside your expertise—web development, database management, CI/CD pipelines, or unfamiliar libraries. AI agents excel here because they've seen millions of examples of how

these technologies work. You don't need to become an expert in Docker or GitHub Actions; you need to describe what you want and review what the agent produces.

Code quality and documentation. Research code often accumulates technical debt: missing docstrings, no tests, inconsistent style. Terminal agents can systematically improve existing code—adding type hints, writing tests, generating documentation—in ways that would be tedious to do manually but are straightforward for an agent.

The risks are equally specific. An agent might generate code that appears to work but contains subtle numerical errors: using single precision where you need double, or implementing an algorithm incorrectly in a way that only fails on edge cases. It might install dependencies with security vulnerabilities, or produce "cargo cult" code that mimics patterns without understanding why they're appropriate.[9] The critical lesson from the vibe coding debate applies here: for throwaway scripts and rapid prototyping, high autonomy is fine. For code that underlies published results, you must understand and verify what the agent produces *in detail*.

The Claude Code Ecosystem: MCP and Skills

In the previous sections, we described Claude Code as a terminal agent that reads, writes, and executes code. But Claude Code's power extends beyond its relationship with your local files. It operates within a growing ecosystem of protocols and conventions that let you customize its behavior and connect it to the outside world. Two pieces of this ecosystem deserve detailed attention: the **Model Context Protocol (MCP)** and the **skills system**.

Model Context Protocol, MCP

Recall from Class 2 that LLMs can use *tools*: structured requests for external actions that let a model search the web, query a database, or run a calculation. The problem, until recently, was that every AI application had to build its own custom connector to every external service it wanted to reach. If you wanted Claude to interact with GitHub, someone had to write a GitHub integration for Claude specifically. If you also wanted it to work with Slack, that was a separate integration. And if you then wanted the same capabilities in Cursor, or in ChatGPT, each of those needed their own connectors too. Ten AI tools and twenty services meant up to two hundred unique integrations to build and maintain.

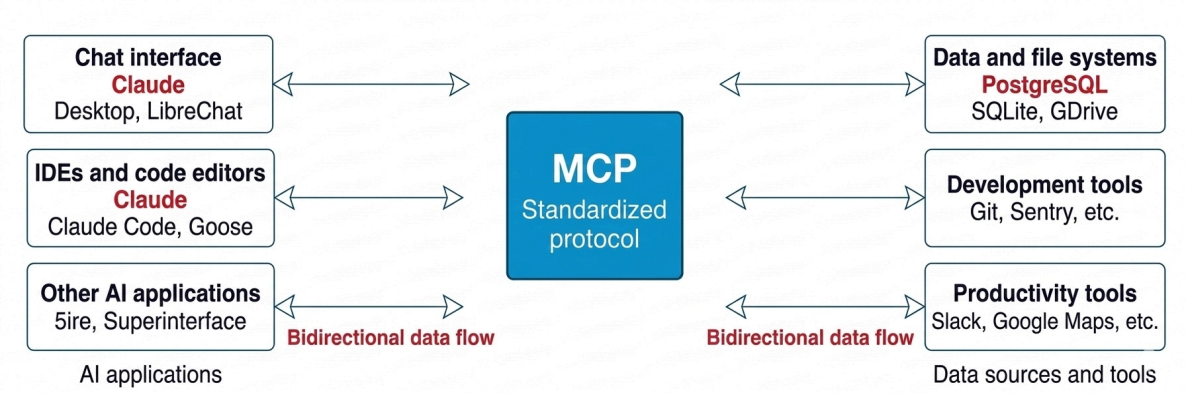
This is sometimes called the **N×M integration problem**, and it's the same problem that USB solved for hardware ages ago. Before USB, every peripheral had its own proprietary cable and port. USB introduced a single standard so that any device could connect to any computer. MCP does the same thing for AI-tool connections.

[9] "Cargo cult" code refers to the practice of including, copying, or implementing code, libraries, or methodologies without understanding how they work ("blind copy-paste"), based on a nearly superstitious belief that because it worked once or elsewhere, it will work again. See: https://en.wikipedia.org/wiki/Cargo_cult_programming

What Is MCP?

The **Model Context Protocol** is an open-source standard, introduced by Anthropic in late 2024,[10] that defines how AI applications communicate with external tools and data sources. The architecture has two sides:

1. An **MCP client** is an AI application that wants to access external capabilities. Claude Code is an MCP client. So are Claude Desktop, Cursor, ChatGPT Desktop, VS Code with Copilot, and many others.
2. An **MCP server** is a lightweight program that exposes a specific tool or service in a standardized way. There are MCP servers for GitHub (managing issues, pull requests, and code search), for web browsers (via Playwright, enabling Claude to navigate and screenshot web pages), for web search (via Brave Search), for databases (PostgreSQL, SQLite), for project management tools (Linear, Jira, Asana), for design tools (Figma), for file storage (Google Drive), and hundreds more. As of early 2026, the ecosystem includes over 200 community-maintained servers.



Derived from MCP documentation image. Edited with Gemini.

The key is **interoperability**: an MCP server built by anyone works with every MCP-compatible client. The GitHub MCP server works in Claude Code, in Cursor, in ChatGPT Desktop, and in any other tool that speaks the protocol. You build (or install) the server once; every AI tool benefits. This is why MCP achieved rapid cross-industry adoption. OpenAI officially adopted it in March 2025, Google DeepMind followed in April 2025, and Microsoft integrated it into VS Code and GitHub Copilot.

How MCP Works in Claude Code

When you add an MCP server to Claude Code, the server registers its available tools (described as JSON schemas) with Claude at the start of each session. Claude then knows what tools are

[10] https://www.anthropic.com/news/model-context-protocol

available and can invoke them as needed during your conversation, just as it would invoke its built-in file-reading or command-execution capabilities.

In practice, adding an MCP server is a one-line terminal command. For example, to connect Claude Code to GitHub:

```
claude mcp add github -e GITHUB_TOKEN=ghp_xxx -- npx -y
@modelcontextprotocol/server-github
```

This command tells Claude Code to launch the GitHub MCP server, pass it your GitHub token for authentication, and register its tools. Once connected, you don't need to name specific tools, you describe what you want in natural language, and Claude selects the right MCP tool automatically. Saying "create a GitHub issue describing this bug" causes Claude to call the `create_issue` tool from the GitHub MCP server with the appropriate parameters.

The practical impact is that Claude Code becomes a central hub that orchestrates your development tools. Instead of switching between your terminal, browser, GitHub web interface, and project management board, you describe your intent and Claude coordinates the work. A developer workflow that might involve six context switches—read the bug report, find the relevant code, search documentation, fix the code, test it, create a pull request—can happen within a single Claude Code conversation.

For detailed documentation, go to https://modelcontextprotocol.io/docs/
See also: *"Connect Claude Code to tools via MCP"* at https://code.claude.com/docs/en/mcp

Why MCP Matters for Research

For engineering researchers, MCP's significance is less about connecting to Slack or Jira and more about the underlying principle: *you can give Claude Code access to any data source or tool by connecting the appropriate MCP server.* Some research-relevant possibilities:

- You could connect Claude Code to a **database server** (PostgreSQL, SQLite) so it can query your experimental data directly, run analyses, and generate plots, all from a natural language request.
- You could connect it to **Google Drive** so it can read your shared lab documents, literature review notes, or datasets without you manually copying content into the conversation.
- You could connect it to a **web search server** so it can look up documentation or recent papers while working on your code.
- You could even write a custom MCP server (the SDK makes this straightforward: under 50 lines of code for a basic server) to expose a domain-specific tool, like a simulation runner or a lab instrument API.

The pattern is always the same: MCP servers extend what Claude Code can *reach*, while the model's intelligence determines *when* and *how* to use those connections.

The Configuration Layer: CLAUDE.md

Before we discuss skills, it helps to understand the simplest way to customize Claude Code's behavior: the **CLAUDE.md** file. This is a plain Markdown file that Claude reads at the start of every session. You can place one globally (in `~/.claude/CLAUDE.md`, which applies to all your projects) and another in any project directory (which applies only when Claude Code is working in that project).

`CLAUDE.md` is where you tell Claude Code about your project's conventions, preferred practices, and important context. For a research project, you might include information like:

> *This project implements a finite volume solver for incompressible Navier-Stokes equations in Python. We use NumPy for array operations and Matplotlib for visualization. Tests are in the `tests/` directory and use pytest. All physical quantities should use SI units. Docstrings should follow NumPy style.*

This is the equivalent of the onboarding document you'd give a new lab member. Claude reads it once at the start of each session, and the conventions inform everything it does. If you tell it to use SI units, it will check for unit consistency when writing or reviewing code. If you specify NumPy-style docstrings, that's what it generates.

A related file, `AGENTS.md`, serves a similar purpose but is recognized by multiple AI tools (Claude Code, Codex CLI, and others), making it useful if you work with more than one agent or if collaborators on your repository use different tools. These are sometimes called the agent's "project memory" files.

Skills: Teaching Claude Code New Expertise

MCP extends what Claude Code can *reach*. The **skills system** extends what Claude Code *knows how to do*. A skill is a folder containing a `SKILL.md` file—a Markdown document with instructions, best practices, and sometimes supporting scripts or templates—that teaches Claude a specific workflow.

The concept is straightforward: imagine you've figured out a reliable process for, say, creating publication-quality PDF figures from your simulation data. The process involves specific Matplotlib settings, a particular color palette, font sizes that match your target journal's requirements, and a sequence of steps for handling different plot types. You could explain this process to Claude Code every time you need a figure. Or you could package it as a skill.

A skill's `SKILL.md` file contains YAML frontmatter (a name and description that Claude uses to decide when to load the skill) followed by detailed instructions in Markdown:

```
---
name: publication-figures
description: Create publication-quality figures following
```

```
  the lab's style guide. Use when asked to generate plots,
  charts, or visualizations for papers.
---

# Publication Figures

## Style Requirements
- Use the 'seaborn-v0_8' style with our custom color palette
- Font: Computer Modern (matching LaTeX documents)
- Figure width: 3.5 inches (single column) or 7 inches (double)
...
```

When you work with Claude Code and mention creating a figure, Claude recognizes that the `publication-figures` skill is relevant, loads it, and follows its instructions. The skill's description acts as a trigger: Claude matches your intent to the description and loads the appropriate knowledge only when needed.

This is an example of **progressive disclosure**: Claude doesn't load all skills into its context at once (which would waste tokens). Instead, it reads only the lightweight metadata (names and descriptions) at startup, then loads the full instructions of a skill only when it determines that skill is relevant to your current task.

Skills vs. MCP: Different Roles

It's worth being precise about the distinction, because MCP and skills sound similar but serve different purposes:

- **MCP** gives Claude Code the ability to *perform actions* it otherwise couldn't—query a database, create a GitHub issue, search the web. MCP is about connecting to external systems.
- **Skills** give Claude Code *knowledge about how to do something well*—your lab's coding standards, a specific workflow for data processing, the requirements for generating figures. Skills are about encoding expertise and best practices.

They complement each other naturally. An MCP server might give Claude Code the ability to query your experimental database; a skill might teach it how your lab structures data queries, which tables are relevant for different experiment types, and how to format the results for your weekly group meeting.

Built-in "Slash" Commands and Skills

Claude Code ships with several **built-in commands** that augment the system's capabilities. E.g.,

`/simplify` reviews your recently changed files for code quality issues and fixes them. It actually spawns three parallel review agents (for code reuse, code quality, and efficiency), aggregates

their findings, and applies corrections. This command is designed to help with **code refactoring and readability**. When you run it against a file or a block of code, it instructs Claude to:

- Remove redundant logic or "dead code."
- Collapse overly complex nested structures into more idiomatic patterns.
- Reduce boilerplate while maintaining the original functionality.
- **Use case:** It is particularly helpful when you have a legacy function that has grown too "curly" or complex over time and you want a cleaner version without changing the logic.

`/batch` orchestrates large-scale changes across a codebase. You describe the change, and it decomposes the work into independent units, creates a plan for your approval, and then spawns one background agent per unit—each working in an isolated Git worktree to avoid conflicts. The `/batch` command is a workflow tool used for **non-interactive task processing**. It allows you to:

- Queue up multiple tasks or files for Claude to process in a single run.
- Run Claude Code in a "headless" or automated fashion where it doesn't wait for user confirmation after every single thought or small change.
- **Use case:** If you need to apply a linting fix or a specific documentation update across twenty different files, `/batch` prevents you from having to manually approve every individual file edit.

Both `/simplify` and `/batch` illustrate a capability called **agent teams**, introduced with Opus 4.6: the ability for Claude Code to spawn multiple sub-agents that work in parallel on different parts of a task. Each sub-agent operates in its own isolated environment (typically a separate Git worktree) so that the agents don't interfere with each other's changes. The main Claude instance acts as an orchestrator: it decomposes the work, dispatches the sub-agents, and aggregates their results. This is a preview of the multi-agent patterns we'll explore in depth later in the course. For now, the key takeaway is that agentic coding is no longer limited to a single agent working sequentially; it can involve coordinated teams working concurrently, which dramatically reduces the wall-clock time for large tasks like codebase-wide refactors or comprehensive code review.

One more built-in command that deserves mention is `/compact`. As you work in a Claude Code session, every file read, command output, and exchange accumulates in the context window. Claude Code does manage this automatically: when the context approaches its limit, it summarizes older content to make room and continues working. But this auto-compaction fires based on token count, not task boundaries, which means it can trigger mid-task and lose details the agent needs (like a specific error message it was debugging). The `/compact` command lets you take control: run it at a natural breakpoint—after finishing a feature, fixing a bug, or before switching to a new task—and Claude summarizes the conversation on your terms. You can even guide what gets preserved, e.g., `/compact focus on the API refactoring decisions and the list of files changed`.

A related command, `/context`, shows you exactly how your tokens are being spent—system prompts, tool definitions, MCP schemas, conversation history—so you can diagnose why a session feels sluggish. For long coding sessions, developing a habit of compacting at task boundaries is one of the most practical things you can learn early.

Copilot CLI and Gemini CLI handle this differently: both auto-compact in the background and do not expose a manual compaction command, which is simpler but gives you less control over what gets preserved.

Built-in Skills and Creating Your Own

Anthropic maintains a public repository of skills at https://github.com/anthropics/skills, covering document creation (Word, PowerPoint, Excel, PDF), frontend design, data analysis, and more. These can be installed as plugins with a single command. And because skills follow the **Agent Skills open standard**, they work not just in Claude Code but also in Claude.ai, Claude Desktop, and the Claude API—you define them once and use them across platforms.

For researchers, the most valuable aspect of the skills system may be the ability to **create custom skills** tailored to your domain. A computational fluid dynamics researcher might create skills for mesh generation workflows, convergence analysis, and post-processing. A materials scientist might create skills for XRD (X-ray diffraction) pattern analysis, phase diagram generation, and DFT (density functional theory) input file preparation.

Creating a skill is as simple as creating a folder with a `SKILL.md` file in your project's `.claude/skills/` directory (for project-specific skills) or in `~/.claude/skills/` (for skills available in all your projects). The skill can include supporting files—Python scripts, templates, reference data—that Claude can read or execute as part of the workflow.

This is a pattern we'll return to throughout the rest of the course: the idea that you can **encode your research expertise into reusable artifacts** that make your AI tools progressively more capable in your specific domain. Skills are one of the most concrete expressions of this idea.

What's Next in your AI-Assisted Coding Journey

Next, you'll get hands-on with specific tools for AI-assisted coding.

1. **GitHub Copilot in VS Code**: You've already activated Copilot through the Student Developer Pack (Class 5). You'll want to explore its chat interface, inline suggestions, and code explanation features in the context of engineering code.
2. **Claude Code**: You'll install Claude Code, configure it for a research project, and walk through an agentic workflow: giving it a task, watching it plan and execute, reviewing its output, and iterating. You'll explore MCP connections and the skills system.

3. **VS Code extensions for AI**: Beyond Copilot, you'll look at the Claude Code extension for VS Code (which embeds Claude Code's terminal capabilities inside the editor) and other AI extensions that bridge the gap between Tier 1 and Tier 3, including Gemini.
4. **AI on Google Colab**: For those who work often in Jupyter notebooks, Colab's AI features offer a low-friction way to get AI assistance during your existing workflow. Explore how these compare to the Jupyter AI tools we learned about in Class 6.

Installing Claude Code

Claude Code requires a paid Anthropic account (Claude Pro at $20/month, at minimum). The free Claude.ai plan does not include Claude Code access. Make sure you have a paid account set up *before* starting installation. **Documentation**: https://code.claude.com/docs/en/setup

On macOS

Open the Terminal app (found in Applications → Utilities, or search for "Terminal" in Spotlight).

Step 1: Install Claude Code. Paste the following command and press Enter:

```
curl -fsSL https://claude.ai/install.sh | bash
```

Wait until you see "Installation complete!"

Step 2: Update your PATH. So your terminal can find the `claude` command from any directory, run:

```
echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.zshrc && source ~/.zshrc
```

(If you use Bash instead of Zsh, replace `.zshrc` with `.bashrc`.)

Step 3: Verify. Close and reopen your terminal, then type:

```
claude --version
```

You should see a version number printed. If you see "command not found," revisit Step 2.

Step 4: Authenticate. Navigate to a project folder (`cd ~/your-project`) and type `claude`.[11] It will ask you to choose a theme: just hit enter, you can change it later. Then it will ask you how to authenticate. The first option, for using your Claude subscription, opens a browser tab where you sign in with your Claude Pro account and authorize Claude Code to use it. You should see:

```
Login successful. Press Enter to continue...
```

[11] For example, you could clone your `tocify` repository from our Class 5 hands-on exercise, then navigate to that folder in the terminal using `cd` to change directory. Since this folder is version controlled with `git`, you have the option to play with Claude's ability to use `git` commands.

You will get some security notes; read them and hit enter again. It will ask you to use Claude Code's terminal setup, choose Yes. You will get a new security question about trusting the code in the folder you're in: hopefully you do, so continue. You're ready to play with Claude!

Alternative Installation (Homebrew): If you use Homebrew, you can install with `brew install claude-code`. Note that Homebrew installations do not auto-update; you'll need to run `brew upgrade claude-code` manually. Go back to Step 2 above.

On Windows

Claude Code on Windows requires **Git for Windows**, which provides Git Bash (used internally by Claude Code to execute commands).

Step 1: Install Git for Windows. Download the installer from git-scm.com/downloads/win and run it, accepting the default options. Make sure the option to add Git to your PATH is selected.

Step 2: Install Claude Code. Open **PowerShell** (not CMD). You can find it by searching "PowerShell" in the Start menu. Paste the following command and press Enter:

```
irm https://claude.ai/install.ps1 | iex
```

Wait for the installation to complete. You do *not* need to run PowerShell as Administrator.

Step 3: Verify. Close and reopen PowerShell, then type:

```
claude --version
```

You should see a version number. If the command is not found, close all terminal windows and try again—your shell needs to pick up the new PATH entry.

Step 4: Authenticate. Navigate to a project folder and type `claude`. Sign in through the browser window that opens, just as on macOS.

Alternative (WSL): If you prefer a Linux-like environment on Windows, you can install Claude Code inside WSL (Windows Subsystem for Linux) using the same `curl` command as macOS. Both WSL 1 and WSL 2 are supported. If you go this route, keep your project files inside the WSL filesystem (e.g., `~/projects/`) rather than on the Windows filesystem (`/mnt/c/...`), which causes significant performance issues.

Install the VS Code Extension

Once Claude Code is installed in your terminal, you can also use it from within VS Code. Search for "Claude Code" in the VS Code Extensions marketplace and install it. This embeds Claude Code's terminal capabilities inside the editor, giving you a Tier 3 agent experience without leaving your IDE—a useful bridge if you're not ready to work exclusively in the terminal.

Troubleshooting

If installation fails, the most common causes are: the PATH not being updated (close and reopen your terminal), an unsupported Node.js version if using the npm method (the native installer above does *not* require Node.js), or network connectivity issues. Running `claude doctor` after installation will auto-detect most configuration problems. If you get stuck, take a screenshot of the error and paste it into Claude.ai or ChatGPT—the irony of using AI to debug your AI tool installation is not lost on anyone, but it works.

Note: if you use conda environments and Claude Code

Since you used the **Native Installer** (`curl ... | bash`), Claude Code was installed as a standalone binary in your user directory, not as a package within your Anaconda environments. Because the installer adds Claude Code to your shell's global PATH (typically in `~/.local/bin`), the command `claude` will work regardless of which conda environment is active.

While Claude Code itself is "outside" of Anaconda, it inherits the environment of the terminal session from which you launch it.

- If you have `(base)` active and you tell Claude to "run my script," it will use the Python version and libraries installed in `base`.
- If you want Claude to work on a specific project, you should activate that project's conda environment first, then run `claude`. This ensures that when Claude tries to run tests or execute code, it has access to the correct dependencies (like `numpy`, `pandas`, or your own research libraries).

The "Claude Code" vs. "Conda" mental model: Conda manages your Python/C++ libraries and compilers, while Claude Code is the "agent" that uses those tools.

Think of it like this: Conda provides the workshop and the tools; Claude is the assistant who walks into that workshop. If you don't activate the right environment, you're essentially leading Claude into a workshop that's missing the tools he needs for your specific project.

Installing GitHub Copilot CLI

GitHub Copilot CLI is included with all Copilot subscriptions, including the free access you get through the **Student Developer Pack** you set up in Class 5. There is no additional subscription cost. **Documentation**: https://docs.github.com/en/copilot/how-tos/copilot-cli/cli-getting-started

On macOS

Open Terminal and run one of the following:

Option A (install script — recommended):

```
curl -fsSL https://gh.io/copilot-install | bash
```

This downloads and installs a self-contained native binary. No Node.js is required (despite what the documentation says, which is almost certainly an oversight from recent updates).

Option B (Homebrew):

```
brew install copilot-cli
```

Both methods auto-update. After installation, close and reopen your terminal, then verify:

```
copilot --version
```

On Windows

Option A (WinGet — recommended):

Open PowerShell and run:

```
winget install GitHub.Copilot
```

This installs a native binary. No Node.js required.

Option B (npm): If you prefer npm, you will need Node.js 22 or later:

```
npm install -g @github/copilot
```

After installation, close and reopen PowerShell, then verify:

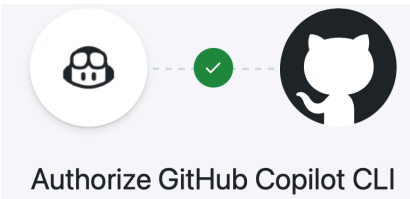
```
copilot --version
```

Authenticate

Navigate to a project folder and start an interactive session:

```
copilot
```

On first launch, enter `/login` and follow the on-screen prompts to authenticate with your GitHub account; you will need to enter a verification code on the URL https://github.com/login/device. You should then see the image below, with a list of orgs that you belong to, scroll down and authorize. This is a one-time step, Copilot CLI remembers your credentials for future sessions.



On the terminal, Copilot CLI should now be active; you will likely see this notice:

```
No copilot instructions found. Run /init to generate a
copilot-instructions.md file for this project.
```

More on customizing your agents below.

Quick Test

Once authenticated, try:

```
Give me an overview of this project.
```

Copilot will analyze the files in your current directory and respond. You can also use Plan mode (press Shift+Tab) to have Copilot outline an implementation plan before making any changes, or pass a one-off prompt without entering an interactive session:

```
copilot -p "Explain what this repository does"
```

Choosing Between Claude Code and Copilot CLI

You now have access to two Tier 3 terminal agents. They are not mutually exclusive—many developers use both. As a rough guide: Claude Code (powered by Opus 4.6) currently leads on deep reasoning, complex multi-file refactors, and has the more mature ecosystem (MCP, skills, agent teams). Copilot CLI has the advantage of zero additional cost through your student subscription, built-in GitHub integration (issues, pull requests, and code search work out of the box), and the ability to choose from models across all three major providers (Claude, GPT, and Gemini). Try both and develop your own sense of when each is most useful.

Watch

Claude Code - Full Tutorial for Beginners

Customizing Your Agent: Instruction Files

You've now installed two terminal agents. Before you start using them on real work, there's one more step that will dramatically improve the quality of their output: **telling them about your project.**

Out of the box, Claude Code and Copilot CLI know nothing about your research, your coding conventions, your preferred libraries, or the structure of your project. They'll still produce useful code, but it will be generic, the kind of code anyone might write for any project. **Instruction files** change this. They're plain Markdown documents that the agent reads at the start of every

session, giving it persistent context about who you are, what you're working on, and how you want things done. Think of them as the onboarding document you'd write for a new collaborator joining your lab.

Where Instruction Files Live

Both agents look for instruction files in specific locations. The system is layered: global files set your personal defaults, and project-level files add context for a specific repository.

Claude Code reads:

- `~/.claude/CLAUDE.md` — your global instructions, applied to every project
- `CLAUDE.md` in your project's root directory — project-specific instructions

Copilot CLI reads:

- `~/.copilot/copilot-instructions.md` — your global instructions
- `.github/copilot-instructions.md` — repo-level instructions
- `.github/instructions/**/*.instructions.md` — scoped instructions for specific parts of the project

Both agents recognize:

- `AGENTS.md` in your project's root directory — a shared instruction file that works across Claude Code, Copilot CLI, Codex CLI, and other tools

This last point is worth emphasizing. If you collaborate with others who use different tools, or if you use both Claude Code and Copilot CLI yourself, `AGENTS.md` is the most portable choice. Write your project instructions once, and every agent that enters the repository will read them.

What to Include

A good instruction file answers the questions that a capable but uninformed collaborator would need answered before contributing to your project. For a research codebase, consider including:

What the project does. A brief description—one or two sentences—of the research problem and the computational approach. This helps the agent understand *why* the code exists, not just what it does.

Key technologies and libraries. Which language, which major libraries, which version constraints matter. If you rely on NumPy, SciPy, and Matplotlib, say so. If you're using a specific framework (FEniCS, OpenFOAM wrappers, PyTorch), the agent should know.

Project structure. Where the source code lives, where tests go, where data files are stored. If you have conventions like "all plotting code goes in `src/visualization/`," state them.

Coding conventions. Docstring style (NumPy, Google, etc.), naming conventions, whether you prefer type hints, how you handle units, any formatting tools you use (Black, Ruff, etc.).

How to run things. The commands for running tests, building documentation, or executing simulations. If there's a Makefile or a specific entry point, mention it.

A Concrete Example

Here's what an `AGENTS.md` file might look like for a graduate student working on computational heat transfer:

```
# Project: Conjugate Heat Transfer Solver

This project implements a finite volume solver for conjugate heat
transfer problems in Python. It couples a fluid solver (incompressible
Navier-Stokes) with a solid conduction solver across shared interfaces.

## Tech Stack
- Python 3.11+
- NumPy and SciPy for linear algebra and sparse solvers
- Matplotlib for visualization
- pytest for testing

## Project Structure
- `src/fluid/` — fluid domain solver
- `src/solid/` — solid domain solver
- `src/coupling/` — interface coupling algorithms
- `tests/` — test suite (run with `pytest`)
- `examples/` — example cases with expected results
- `docs/` — documentation source files

## Conventions
- All physical quantities use SI units
- Docstrings follow NumPy style
- Variable names follow the notation in Chen & Sparrow (1983)
- Type hints on all public functions
- No print statements in library code; use Python's logging module

## Important Notes
- The coupling interface is order-sensitive: always update fluid
  before solid in each iteration
- Sparse matrices use CSR format throughout; do not convert to dense
```

This file is 30 lines long, takes ten minutes to write, and will be read by the agent at the start of every session for as long as you work on this project. The return on that ten-minute investment is substantial: the agent will generate code that uses your naming conventions, respects your unit system, follows your project structure, and runs your tests correctly, without being reminded each time.

Start Simple, Iterate

You don't need to write the perfect instruction file on day one. Start with the basics — what the project does, what language and libraries it uses, and where the tests are. As you work with the agent and notice it making incorrect assumptions (wrong docstring style, wrong directory for new files, forgetting to use type hints), add those corrections to the instruction file. Over time, the file becomes a living record of your project's conventions, useful not only for AI agents but for human collaborators too.