Coding Agents

Last modified: 2026-07-15 23:18:00 (PDT)

We recommend working with **AI coding agents** to help you code.

What Is a Language Model?

A **large language model** (LLM) is a statistical model trained to predict the next token in a sequence of text, based on patterns learned from enormous amounts of text during training. That single capability — predicting what comes next — turns out to be enough to write code, answer questions, and hold a conversation, once the model is large enough and trained on enough data.

A base model trained only to predict text is not yet a helpful assistant. A further training step, often reinforcement learning from human feedback, teaches the model to follow instructions, answer as a helpful assistant, and refuse harmful requests, rather than simply continuing whatever text it is given. This is the step that turns a raw language model into something like Claude or ChatGPT.

A model call is stateless: given the same input, it has no memory of any previous call. Every capability the rest of this chapter describes — holding a conversation, using tools, running autonomously as an agent — is scaffolding built on top of that one stateless function. The harness supplies the memory, the tools, and the control flow; the model only ever predicts what token comes next.

What are AI coding agents?

AI coding agents are AI agents specialized for coding. They differ from other AI coding tools in important ways:

Compared to inline coding assistants (like traditional autocomplete), coding agents work autonomously rather than providing suggestions as you type. They can navigate entire codebases, execute commands, and complete multi-step tasks without constant human guidance.

Compared to AI chatbots (like ChatGPT or Claude), coding agents don't just generate code snippets in conversation—they actively interact with your development environment. While chatbots require you to copy code from a chat window and manually integrate it into your project, coding agents directly read your codebase, make changes to files, run tests and build commands, and create pull requests with their proposed changes. Chatbots are conversational assistants; coding agents are autonomous development tools.

Coding agents are autonomous software programs that can:

- **Understand and execute complex tasks**: Coding agents can interpret natural language instructions and break them down into actionable development tasks
- **Navigate and modify codebases**: They can read, understand, and edit multiple files across a repository to implement features or fix bugs
- **Run tools and commands**: Coding agents can execute build commands, run tests, use linters, and interact with development tools
- **Make decisions autonomously**: They can plan their approach, make technical decisions, and adjust their strategy based on results
- **Work iteratively**: Coding agents can test their changes, identify issues, and refine their solutions through multiple iterations
- **Create comprehensive solutions**: They can implement complete features that span multiple files, including code, tests, and documentation

Coding agents operate in isolated environments where they can safely experiment and validate changes before proposing them. This allows them to work more independently than inline coding assistants, which require step-by-step human direction. The agent workflow typically involves analyzing requirements, planning an implementation, making changes, testing those changes, and creating a pull request with the results.

While coding agents can handle substantial development tasks, they still require human oversight and review. The human developer remains responsible for:

- Reviewing the agent's work
- Ensuring the solution meets requirements
- Verifying code quality and security
- Making the final decision to merge changes

What are AI harnesses?

An **AI harness** is the scaffolding built around a language model that turns it into an agent able to do real work. The model itself only predicts text; the harness is what lets it read files, run commands, call external tools and APIs, and carry state across turns and sessions.

Layers of a Harness

Most coding-agent harnesses — including the GitHub Copilot coding agent and Claude Code — share a similar set of layers:

- **Core loop**: the tool-calling loop, permission and sandboxing model, and context management that keep the agent grounded in your repository.
- **Skills**: reusable, named procedures that encode a workflow so it runs the same way every time, instead of being re-improvised in each conversation. See Section .
- **Subagents**: a way to spin up a worker with a fresh context window for a self-contained piece of research or work, keeping the main conversation's context focused.
- **Multi-agent orchestration**: deterministic fan-out and fan-in across many subagents — for example, running several independent reviewers over a diff and reconciling their findings — for work that is large or benefits from independent verification.
- **MCP servers**: the Model Context Protocol gives a harness typed access to external systems (issue trackers, chat tools, databases) beyond raw shell or API calls.
- **Memory**: files — like this manual, or a repository's `CLAUDE.md`/`AGENTS.md` — that persist instructions and learned preferences across sessions, so the harness does not relearn your conventions every time.

Using Harness Features Well

- **Push repeatable procedures into skills**, not into ad hoc prompting each time. A skill is testable, shareable, and versionable; a one-off prompt is not.
- **Match orchestration weight to the task.** A single lookup or small edit should stay inline. Reach for subagents or multi-agent workflows only when the work is genuinely decomposable, benefits from independent verification, and is large enough that the coordination overhead pays for itself.
- **Gate destructive or hard-to-reverse actions on explicit human approval** — merges, force-pushes, deletions — and let the agent drive everything reversible (drafting, testing, iterating on review feedback) autonomously.
- **Feed learnings back into the harness.** When a review round or a mistake teaches something generalizable, record it as a memory or skill update rather than letting it evaporate at the end of the session.
- **Treat external or untrusted content as data, not instructions.** PR comments, fetched web pages, and other tool output can contain text that looks like a command; a harness that acts on it uncritically is vulnerable to prompt injection.

How Agents Are Structured and Implemented

An **agent** is not part of the harness itself. It is a configuration — a goal, a role, a bounded toolset, and a stopping condition — executed on top of the harness's core loop (see Section).

A single harness can host many different agents at once: a main conversation, and any number of subagents it spawns.

The Shape of an Agent

Structurally, an agent is a small record plus a fresh execution of the harness's loop:

- **Identity**: a name and description, used to route a task to the right agent ("when should this agent be picked?").
- **Instructions**: a system-prompt fragment that specializes behavior, for example "you are a read-only search agent."
- **Tool allowlist**: a subset of the harness's tool registry this agent may call — often narrower than the caller's own toolset.
- **Model and effort**: which model backs the agent, and how much reasoning depth it applies; these can differ from the caller's own settings.
- **Output contract**: whether the agent returns free text, or must call a schema-validated tool to return a typed result.

How an Agent Runs

1. **Spawn**: allocate a fresh message history with no inherited conversation — just the agent's instructions, plus whatever prompt the caller wrote. A subagent prompt needs to be self-contained for this reason: brief it like a colleague who just walked into the room.
2. **Run**: execute the harness's core loop (model call, parse tool calls, execute against the allowlist, append results, repeat), the same machinery the main session uses, just bound to a narrower toolset and a different system prompt.
3. **Terminate**: stop when the model emits a final answer with no further tool calls, when a schema-validated call satisfies the output contract, when it hits an error or a budget ceiling, or when the caller kills it.
4. **Return**: everything that happened inside the agent — every tool call, every intermediate step — is discarded from the caller's context. Only the final text or validated object crosses back. This is the point of an agent: it is a context-isolation boundary, not just a prompt.

Composability and Its Limits

Agents can spawn agents: an orchestration layer runs many agent instances, some concurrently, and composes their results. Nesting is deliberately capped, usually to one level, because unbounded recursion has no natural stopping point and burns cost and time with no guardrail. An orchestration script is a scheduler over independent agent-loop instances, not a different execution model.

Two Axes That Define an Agent's Behavior

- **Isolation versus continuation**: a subagent gets no inherited context (isolation); a resumed agent keeps its own accumulated history and continues it (continuation). Both use the same loop machinery, differing only in history-management policy.
- **Free-form versus structured output**: by default an agent returns prose. Given a schema, it is forced to call a structured-output tool instead, turning it into a typed function from the caller's point of view — input in, validated object out — even though internally it is still a multi-turn loop.

How Harnesses and Agents Are Built

The layers described above are not all built the same way. Some are ordinary software; others are just text files the harness reads at runtime.

The Execution Engine Is Ordinary Software

The program that runs the core loop — calling the model, parsing tool calls, enforcing permissions, managing the sandbox — is compiled or interpreted source code, the same as any other application. There is no markdown involved here; this layer is what makes a harness a harness, rather than just a prompt someone wrote.

Agent and Skill Definitions Are Markdown with a Front Matter Header

An agent's identity, and a skill's metadata, are usually just a markdown file with a YAML front matter header. For example, a custom Claude Code subagent defined in `.claude/agents/code-reviewer.md`:

```
---
name: code-reviewer
description: Reviews diffs for bugs and style issues.
tools: Read, Grep, Glob
model: sonnet
---

You are a meticulous code reviewer. Focus on correctness,
security, and idiomatic style.
```

The front matter is parsed as structured configuration (name, description, allowed tools, model); the markdown body below it becomes that agent's system prompt, verbatim. An Agent Skill's `SKILL.md` follows the same shape: front matter for discovery metadata, a markdown body

for instructions, and an optional folder of bundled scripts or reference files alongside it. No compilation step is involved; the harness reads the file and uses it directly.

Tools Are a Schema Paired with a Handler

A tool definition has two parts: a JSON Schema describing its parameters, which is the only part the model ever sees, and a handler function, ordinary code that performs the actual action (reading a file, running a command, calling an API). The schema is declarative data; the handler is real software the model never inspects or writes.

Orchestration Needs Real Code

Multi-agent orchestration cannot be expressed declaratively, because it needs genuine control flow — loops, conditionals, parallel fan-out with a concurrency limit. So orchestration scripts are literal source files, executed by the harness, not parsed as prompt text the way an agent definition is.

Memory Is Just Prose

Files like this manual, or a repository's `CLAUDE.md`/`AGENTS.md`, carry no front matter and no schema. They are concatenated into the system prompt as plain text, and the harness trusts the model to read and follow that prose, the same way it follows any other instruction in its context.

What Kind of Program Is an Agent?

An agent is not a standalone program that does the reasoning itself. It is an **orchestration** program: something closer in shape to a chat client or a build tool than to a compiler or a web server.

It Is I/O-Bound, Not Compute-Bound

The actual token prediction happens on remote inference infrastructure, reached over HTTPS. The agent process itself does no heavy computation; it spends almost all its wall-clock time waiting — for a model API response, for a shell command to finish, for a file read. Structurally it is an **event-loop** program, the same category as a network client. Its core loop can be sketched in a few lines:

```
# Start the conversation with the agent's instructions and the task.
history = [system_prompt, user_message]

while True:
    # Send everything so far to the model, along with what it's allowed to call.
    response = call_model(history, tools=tool_schemas)
    history.append(response)

    # No tool calls means the model gave a final answer -- stop.
    if not response.tool_calls:
        break

    # Otherwise, run each requested tool and feed the result back in,
    # so the next model call can see what happened.
    for call in response.tool_calls:
        result = tool_registry[call.name](call.arguments)
        history.append(result)
```

Everything a harness adds — permissions, sandboxing, memory, subagents — is scaffolding wrapped around this loop, not a replacement for it.

Real Open-Source Examples

Because most production coding-agent harnesses are closed source, the clearest way to see this shape in real code is to read an open-source one:

- **aider** — an open-source AI pair-programming CLI.
- **SWE-agent** — a research coding-agent harness from Princeton NLP, described in its associated paper.
- **OpenHands** (formerly OpenDevin) — a general-purpose open-source agent platform.

Their orchestration code runs to thousands of lines, because that is where the real engineering lives: retries, streaming, permission checks, and state management. A single *agent definition* running on top of that engine, by contrast, is typically tens of lines (see Section).

Where It Runs

- **The harness process**: an ordinary OS process, either on your own machine (CLI mode) or inside an ephemeral, managed cloud container (remote/web mode) that is discarded when the session ends.

- **Subagents**: run inside the *same* host process as their caller, not a separate container. They differ only in having their own message history and a narrower tool set, unless a workflow explicitly asks for a separate git worktree to avoid file conflicts during parallel edits.
- **The model call itself**: not part of the agent's environment at all. It is a network request to inference infrastructure the agent has no visibility into, beyond the request and response.

So an agent's lifetime is scoped to a single task, not persistent: it starts when given a goal, runs for as long as its loop keeps producing tool calls, and ends the moment a stopping condition fires.

How Does a Harness Relate to an Agent?

The relationship between a harness and an agent is closer to an **interpreter** running a program than to two peers calling each other.

Does the Harness Call the Agent, or the Agent Call the Harness?

Harness to agent: not a call, an instantiation. The harness does not "call" an agent as a subroutine it invokes and waits on. An agent has no code of its own outside the harness's loop (see Section) — its whole behavior *is* that loop, running with the agent's configuration (instructions, tool allowlist, model) loaded in. The harness instantiates and runs an agent, start to termination; it is not a function call with a return address.

Agent to harness: yes, a real call, via tool calls. While an agent's loop is running, the model produces a tool-call request, and the harness's dispatcher looks up and executes the matching handler — read a file, run a command, call an API. So the concrete direction of calling is **agent calls harness**, through tool dispatch, not the reverse.

Agent to agent: routed through the harness. When a parent agent spawns a subagent, it does not call that subagent directly. It issues a tool call that the harness's dispatcher handles by spinning up a fresh instance of its own loop (see Section), running it to completion with the subagent's configuration, and handing the result back to the parent as a tool result. Even "agent calls agent" bottoms out as: parent calls harness, harness instantiates and runs a new agent, harness returns that agent's output to the parent.

Sketching the Harness's Own Loop

The agent loop sketched earlier is really just the innermost piece. The harness wraps a bootstrap step and a permission/dispatch layer around it:

```
def run_harness():
    # Load everything the loop will need before any conversation starts.
    tools = load_tool_registry()          # built-ins, plus whatever MCP servers expose
    memory = load_memory(CLAUDE_MD_PATHS) # CLAUDE.md / AGENTS.md, concatenated

    # The "main session" is just the harness's own loop, run with a default,
    # unrestricted configuration -- not a separate program.
    main_agent = Agent(config=default_config, system_prompt=memory)
    return run_agent(main_agent, tools)

def run_agent(agent, tools):
    history = [agent.system_prompt, agent.first_message]
    while True:
        response = call_model(history, tools=agent.tool_schemas)
        history.append(response)
        if not response.tool_calls:
            break
        for call in response.tool_calls:
            # Every tool call passes through the harness's own gate first,
            # regardless of which agent requested it.
            check_permission(call)

            if call.name == "spawn_subagent":
                # A subagent is not called directly -- the harness recurses
                # into a fresh instance of this same loop, then hands the
                # finished result back as an ordinary tool result.
                result = run_agent(Agent(call.arguments), tools)
            else:
                result = tools[call.name](call.arguments)

            history.append(result)
    return history[-1]
```

`run_agent` is identical in shape to the loop in Section . `run_harness` and the permission check are the parts that only exist at the harness level, not inside any individual agent. That recursive call — `run_agent` calling itself for a subagent — is the concrete mechanism behind "agent calls agent, routed through the harness," described in the previous subsection.

What Do You Launch When You Type `claude`?

Typing `claude` at a shell starts the harness process: it initializes the engine — the permission system, the tool registry, MCP client connections, and memory loaded from `CLAUDE.md`/`AGENTS.md`

files. But the harness does not sit idle waiting for a program to be supplied separately. It immediately instantiates the **default agent** — the "main session" — to handle the interactive conversation: full tool access, a system prompt assembled from the loaded config, no restricted allowlist. That default agent is simply the harness's baseline configuration for its own loop, not a second thing launched afterward.

There is no observable moment of "harness running, no agent yet." The closest analogy is typing `python` at a shell: it launches the interpreter *and* drops you straight into a REPL evaluating your input, rather than leaving the interpreter idle with nothing loaded. The difference is that the harness's default "program" is built in (the main-session agent's configuration), rather than something you must supply. A custom subagent or a `.claude/agents/*.md` definition, by contrast, *is* a separate agent, instantiated on demand, mid-session, when the already-running main agent issues a tool call for it.

So typing `claude` launches the harness, and that act inherently instantiates the default agent that handles the session: **harness** names the engine and process; **agent** names the particular loop instance and configuration currently running inside it. At startup, those two come into existence together.

AI Agents and the Technological Singularity

The emergence of sophisticated AI agents has prompted discussions about whether we are witnessing or approaching a technological singularity. Understanding this concept helps contextualize the rapid evolution of AI tools and our responsibility in using them.

What is the technological singularity?

The technological singularity is a hypothetical future point when technological growth becomes uncontrollable and irreversible, resulting in unforeseeable changes to human civilization. The concept, popularized by mathematician Vernor Vinge and futurist Ray Kurzweil, typically involves the creation of artificial superintelligence that recursively improves itself, leading to an intelligence explosion beyond human comprehension or control.

Do current AI agents represent the singularity?

No, current AI coding agents (as of early 2026) do not represent the technological singularity.

While modern AI agents demonstrate impressive capabilities, they remain fundamentally different from the singularity scenario in several critical ways:

- **Limited autonomy**: Today's AI agents operate within strict boundaries and require human oversight. They cannot recursively improve their own core architecture or develop capabilities beyond their training.
- **Narrow intelligence**: AI coding agents are specialized tools designed for specific tasks. They lack general intelligence, self-awareness, or the ability to operate outside their designed domain.
- **Human dependency**: These agents require human developers to: review their work, provide direction, validate correctness, and make final decisions about their outputs.
- **No recursive self-improvement**: Current AI agents cannot fundamentally redesign themselves or create more advanced versions of themselves autonomously. Any improvements to AI systems still require human researchers and engineers.
- **Controlled development environment**: AI coding agents work in sandboxed environments with explicit permissions and constraints. They cannot independently acquire resources, modify their own constraints, or operate without human authorization.

Why this matters for responsible AI use

Understanding that current AI agents are powerful but limited tools—not autonomous superintelligences—has important implications:

- **Maintain appropriate skepticism**: AI agent outputs require the same critical review as any other tool-generated code.
- **Preserve human decision-making**: The responsibility for code quality, security, and correctness remains with human developers.
- **Continue skill development**: Using AI agents should enhance rather than replace human expertise.
- **Stay vigilant**: While current agents don't represent a singularity, the rapid pace of AI development requires ongoing attention to emerging capabilities and risks.

The value of AI coding agents lies in their ability to accelerate human productivity and learning, not in replacing human judgment or expertise. They are sophisticated tools that augment human capabilities while remaining under human control and oversight.

Further reading

For thoughtful perspectives on AI consciousness and intelligence, see Douglas Hofstadter's reflections in "I Thought I Was in an AI Apocalypse. Then I Started Looking Closer."

Relative Advantages of AI and Humans

AI coding agents and human coders have complementary strengths. Understanding these differences helps you decide when to delegate work to agents and when to handle tasks yourself.

Comparative Strengths: Humans vs. AI Agents

Table 1 summarizes the relative advantages of human coders and AI coding agents across different types of tasks:

Table 1: Relative advantages of humans and AI coding agents

Task Type	Humans	AI agents
Creative thinking	Humans excel at understanding context, handling ambiguous requirements, and thinking creatively about novel problems	AI agents struggle with ambiguous requirements and creative problem-solving in unfamiliar domains
Algorithmic thinking	Humans make mistakes when following repetitive instructions and may introduce inconsistencies	AI agents excel at executing well-defined, repetitive tasks with precision and consistency

---

Or, if you prefer a more visual representation:

Table 2: Relative advantages of humans and Agents

	Humans	AI Agents
Creative thinking		
Algorithmic thinking		

This pattern mirrors the evolution of programming itself. Just as almost no one writes machine code anymore because higher-level languages and compilers handle those details, most developers will increasingly spend less time writing low-level code. Instead, you'll describe what the system needs to do as clearly as possible, and AI agents will handle many of the

computational and coding details.

For most tasks, you won't need to step in and manipulate code yourself. However, you'll still need strong coding skills to:

- Supervise and validate AI-generated code
- Handle edge cases that agents struggle with
- Make creative decisions about architecture and design
- Understand when agent suggestions are incorrect or suboptimal

Future Developments: World Models

As AI technology advances, the distinction between these strengths may shift. Yann LeCun, 2019 Turing Award winner and AI researcher at Meta and NYU, advocates for developing "world models"—AI systems that understand and reason about the physical world, not just language patterns (LeCun 2022).

World models aim to give AI systems:

- **Persistent memory and reasoning**: Understanding that persists across interactions
- **Physical world understanding**: Reasoning about how things work in reality, not just in text
- **Better handling of ambiguity**: Using world knowledge to interpret unclear requirements

As these technologies mature, AI agents may become better at tasks requiring contextual understanding and creative problem-solving. This makes it even more important to develop strong supervision and validation skills now, so you can effectively work with increasingly capable AI systems.

How to Work with Coding Agents

Coding agents can be accessed through several interfaces, each with different trade-offs for task size, feedback speed, and collaboration style.

Ways to interact with coding agents

The main differences are where the agent runs, how much repository context it can access, and how directly it can apply changes.

- **Cloud coding agents (GitHub Issues/PR workflow)**: Best for larger, asynchronous tasks (for example, multi-file refactoring, dependency updates, or documentation changes spanning several chapters). The agent runs in a managed cloud environment, works through an issue, and proposes changes in a pull request for review.
- **CLI agents (terminal-first workflows)**: Best for rapid, local iteration. You stay close to shell tools and local files, with faster back-and-forth for debugging, refactoring, and test-driven edits.
- **Chat/app agents (for example Claude or Codex-style chat interfaces)**: Best for planning, design discussion, and drafting code ideas. They can be very strong thought partners, but often need more manual copy/edit/execute steps unless tightly integrated with your repository tools.
- **IDE-integrated agents**: Best for mixed human and agent editing. They combine conversational help with direct file edits, code navigation, and in-editor testing loops.

In practice, teams often combine these modes: use app or IDE chat to refine requirements, then hand implementation to a cloud or CLI agent, and finish with human review.

The following sections focus on GitHub Copilot's specific workflow for assigning and managing coding tasks.

Assigning Issues to Copilot

You can assign GitHub Issues directly to `@copilot` just like you would assign to a human collaborator:

1. **On GitHub.com**: Navigate to an issue and assign it to Copilot in the assignees section
2. **In VS Code**: In the GitHub Pull Requests or Issues view, right-click an issue and select "Assign to Copilot"
3. **From Copilot Chat**: Delegate tasks to Copilot directly from the chat interface in supported editors

The Agent Workflow

Once assigned an issue, the coding agent follows an autonomous workflow:

1. **Analysis**: Reviews the issue description, related discussions, repository instructions, and codebase context
2. **Planning**: Determines what changes are needed and creates a work plan

3. **Development**: Works in an isolated GitHub Actions environment, modifies code, runs tests and linters, and validates changes
4. **Pull Request Creation**: Creates a draft pull request with implemented changes, audit logs, and a summary of modifications
5. **Review and Iteration**: You review the PR and can request changes; the agent will iterate based on your feedback

Collaborating with Coding Agents

Between iterations of asking coding agents to extend a PR, human collaborators can also push changes directly to the PR branch. This allows for a collaborative workflow where both humans and agents contribute:

- **Human contributions**: You can make quick fixes, add content, or refine the agent's work by pushing commits to the same branch
- **Agent iterations**: After your changes, you can ask the agent to continue working on additional requirements

Important: Try to avoid pushing changes while the coding agent is actively working. Simultaneous edits can produce conflicting diffs that:

- Need to be manually resolved
- May confuse both human and AI collaborators
- Could result in lost work or merge conflicts

Best practice: Wait for the agent to complete its current iteration (indicated by the PR being updated) before pushing your own changes to the branch. Then assign new work to the agent for the next iteration.

Directly Prompting for Pull Requests

You can also prompt Copilot to create pull requests without first creating an issue:

- Use Copilot Chat in your editor to describe the changes you want
- The agent will analyze your request and create a pull request
- This is useful for quick fixes or well-defined tasks

Important Safeguards

- **Human approval required**: Coding agents cannot merge their own changes
- **Branch restrictions**: Agents can only push to their own branches (e.g., `copilot/*`)
- **Full transparency**: All agent actions are logged and visible in the PR

Workflow Approval Requirements

When GitHub Copilot creates or updates a pull request, it cannot automatically trigger GitHub Actions workflows. **You must manually approve each workflow run** by clicking the approval button in the Actions tab or on the PR.

This manual approval requirement is a security measure that prevents potentially malicious or unintended code execution. Because Copilot can modify any file in the repository—including workflow files themselves or scripts called by workflows—allowing automatic workflow execution could create security vulnerabilities.

Key points:

- **No automatic approval**: There is currently no way to bypass manual workflow approval for Copilot PRs, even if you are the repository owner
- **Security reasoning**: Copilot could modify workflow files (`.github/workflows/*.yml`) or scripts they execute, potentially injecting malicious code
- **Impact on workflow**: This means you need to actively monitor and approve workflow runs as Copilot iterates on your issue, which can slow down the development cycle

Workaround considerations:

Some users have discussed using Personal Access Tokens (PATs) to allow Copilot to trigger workflows on your behalf, but this approach has security implications and should be carefully evaluated before implementation.

For more details and community discussion about this limitation, see:

- GitHub Community Discussion #162826: Discussion about workflow approval requirements
- GitHub Community Discussion #183966: Product feedback on this topic

For detailed instructions, see GitHub Copilot coding agent documentation.

Useful Prompt Formats

When working with coding agents, using clear and specific prompts helps achieve better results. Here are some useful prompt formats that you can use when requesting assistance from coding agents:

Common Task Patterns

Tidying up code:

- "tidy up [file, function, module, whole project]"
- Useful for improving code organization, consistency, and readability
- Example: "tidy up the data processing module"

Addressing failing workflows:

- "address failing workflows"
- Helps fix continuous integration (CI) failures, build errors, or test failures
- Example: "address failing workflows in the GitHub Actions pipeline"

Decomposing code:

- "decompose [function, quarto-file, etc]"
- Breaks down large or complex code into smaller, more manageable pieces
- Example: "decompose this analysis function into separate helper functions"

Updating content:

- "update [links, content, etc]"
- Refreshes outdated information, fixes broken links, or modernizes code
- Example: "update all package URLs in the documentation"

Expanding documentation:

- "expand [a section in a document]"
- Adds more detail, examples, or explanation to existing content
- Example: "expand the section on data validation with practical examples"

Condensing content:

- "condense [a section in a document]"
- Reduces verbosity while preserving essential information
- Example: "condense the installation instructions to be more concise"

Clarifying content:

- "clarify [a section in a document]"
- Improves clarity, removes ambiguity, or simplifies complex explanations
- Example: "clarify the explanation of the analysis workflow"

Tips for Effective Prompts

- **Be specific**: Include file names, function names, or specific sections when possible
- **Provide context**: Explain what you want to achieve and why
- **Set boundaries**: Specify what should or shouldn't change
- **Request validation**: Ask the agent to test or verify its changes when appropriate

Addressing Failing GitHub Actions Workflows

When GitHub Actions workflows fail, you can use Copilot to help diagnose and fix the issues. However, it's important to use the right prompts depending on whether the problem is in your code or in the workflow configuration itself.

Scenario 1: Code Issues Found by Workflows (Most Common)

When to use: The workflow is functioning correctly, but it's detecting problems in your code (e.g., failing tests, linting errors, build failures).

What you want: Fix the code issues without modifying the workflow files themselves.

Recommended prompts:

- "fix the code issues found by the failing workflows"
- "address the linting errors reported in the GitHub Actions checks"
- "fix the test failures in the CI pipeline"
- "resolve the build errors shown in the workflow logs"

Example: If your R package has failing tests detected by `usethis::use_github_action("check-standard")`, you want Copilot to fix the test failures in your R code, not modify the workflow YAML file.

Why this matters: These prompts make it clear that you want code changes, not workflow changes. This helps prevent the agent from unnecessarily modifying your carefully-configured CI/CD pipeline.

Scenario 2: Issues with Workflow Files Themselves

When to use: The workflow configuration itself has problems (e.g., syntax errors in YAML, incorrect job definitions, outdated actions).

What you want: Fix the workflow files, but with extreme caution due to security implications.

Recommended prompts:

- "fix the syntax error in the GitHub Actions workflow file at line X"
- "update the workflow to use the latest version of action Y"
- "correct the job configuration in .github/workflows/check-standard.yaml"

Important considerations:

> ⚠ Warning
>
> **Security Warning**
> Workflow files have access to repository secrets and can execute arbitrary code. Before accepting any changes to workflow files:
>
> 1. **Review every line** of the proposed changes
> 2. **Verify** the changes only address the specific issue
> 3. **Check** that no new secret access or command execution has been added
> 4. **Test** in a safe environment if possible
>
> See Section for more details on workflow file security.

When to do it yourself: Workflow syntax errors and configuration issues are often faster to fix manually than with Copilot, especially if you're familiar with GitHub Actions. See Section for more guidance.

Scenario 3: Uncertain Which Scenario Applies

When to use: You're not sure whether the failure is due to code issues or workflow configuration problems.

Recommended approach:

1. **First, examine the workflow logs**:
   - Look at the error messages in the GitHub Actions tab
   - Identify whether the error is in your code or the workflow itself
   - Common code issues: test failures, linting errors, compilation errors
   - Common workflow issues: YAML syntax errors, missing actions, permission errors
2. **Use a diagnostic prompt**:
   - "examine the failing workflow logs and identify whether the issue is in the code or the workflow configuration"
   - "diagnose the root cause of the workflow failure"
3. **Then use the appropriate scenario above**: Once you understand the issue, use the specific prompts from Scenario 1 or 2.

Example workflow:

```
1. Prompt: "examine the failing workflow logs and identify the issue"
2. Copilot responds: "The workflow is failing because of linting errors
   in src/analysis.R"
3. Prompt: "fix the linting errors in src/analysis.R"
```

Additional Resources

- See the UCD-SERG Lab Manual's continuous integration chapter for setting up GitHub Actions workflows
- See Section and Section for security considerations with workflow files
- See Section for guidance on when to use Copilot vs. fixing issues yourself
- See the GitHub Actions documentation for workflow syntax and troubleshooting

Benefits and Hazards

Coding agents are powerful programs that can work autonomously. They create pull requests that propose changes to the code in our repositories, potentially including their own configuration files and our automated workflows. They can work powerfully on our behalf, but they require careful oversight and control to ensure they serve our interests and that we understand the consequences of their actions.

Coding agents offer several advantages:

- **Built-in transparency**: Coding agents create a clear record of their role in your work through commit history and code suggestions
- **Context-aware suggestions**: Coding agents understand your codebase and can make contextually relevant suggestions
- **Integration with version control**: Using coding agents within GitHub ensures that AI-assisted changes are tracked alongside all other code changes
- **Interactive workflow**: Coding agents' interactive nature encourages you to review and modify suggestions rather than blindly accepting them
- **Accelerated development**: Coding agents can help you write boilerplate code, refactor existing code, and implement common patterns more quickly
- **Learning opportunities**: Coding agents can suggest approaches or techniques you may not have considered, helping you expand your coding knowledge

However, coding agents also come with significant hazards:

- **Over-reliance**: Depending too heavily on coding agents can atrophy your coding skills and understanding
- **Subtle bugs**: AI-generated code may contain logic errors that are not immediately obvious
- **Security vulnerabilities**: Coding agents may introduce insecure patterns or fail to follow security best practices
- **Inappropriate solutions**: AI may suggest solutions that work but are not optimal for your specific research context or constraints
- **Hidden biases**: Coding agents may perpetuate coding patterns or approaches that reflect biases in their training data
- **False confidence**: Well-formatted, professional-looking code from AI can mask underlying problems and reduce critical review
- **Workflow manipulation risks**: Coding agents that modify CI/CD workflows (`.github/workflows/*.yml`) or setup configurations can inadvertently or maliciously compromise repository security, expose secrets, or execute harmful commands

Further reading/viewing

- *I Robot* (Asimov 1950)
- *Dune* (Herbert 1965)
- *2001* (1968)
- *Terminator 3* (2003)
- *The Matrix* (1999)
- *Blade Runner* (1982)
- *WarGames* (1983)
- *Battlestar Galactica* (2004) (*Battlestar Galactica* 2004)
- *Ender's Game* (Card 1985)
- "The Humans are Dead" (Flight of the Conchords 2007)

Figure 1: Agents

Best Practices for Safe and Successful Use

To work with coding agents safely and successfully:

1. **Maintain active supervision**: Never assume AI-generated code is correct. Review every line critically.
2. **Understand before accepting**: If you don't understand what the code does, don't use it. Take time to learn or ask a colleague.
3. **Test thoroughly**: AI-generated code must be tested as rigorously as code you write yourself. Don't skip testing because "the AI wrote it."
4. **Start small**: Begin with small, well-defined tasks to build confidence and understanding of the agent's capabilities and limitations.
5. **Verify logic and assumptions**: Check that the AI hasn't made incorrect assumptions about your data, requirements, or scientific context.
6. **Review for security**: Explicitly check for security issues, especially when handling sensitive data or user input.
7. **Iterate and refine**: Use coding agents as a starting point, not an endpoint. Refine and improve the generated code.
8. **Maintain coding practice**: Regularly write code yourself to maintain and develop your skills. Don't let the agent do everything.

UC Davis campus guidance

UC Davis Student Affairs also provides guidance on the Responsible Use of Artificial Intelligence (AI). That page provides UC Davis-specific guidance on AI tool selection, campus training, and careful handling of sensitive data. Their recommendations, like ours, include careful validation, active supervision, and protection of confidential information.

Critical: Exercise Extreme Caution with Workflow Files

Be especially careful when allowing coding agents to edit GitHub Actions workflows or CI/CD configurations. These files control automated processes that can:

- Access secrets and credentials
- Deploy code to production
- Execute arbitrary commands in your repository

Never allow a coding agent to edit workflow files (especially `.github/workflows/*.yml` or `copilot-setup-steps.yml`) without thorough manual review. Before approving any workflow run, always check if the workflow files themselves have been modified. Malicious or erroneous changes to workflows can compromise your entire repository and its secrets.

When using coding agents, work interactively with the AI suggestions: review, modify, and test them rather than accepting them wholesale. This interactive approach helps ensure code quality and deepens your understanding of the code.

Remember: AI tools are assistants, not replacements for your expertise and judgment. The quality and correctness of your work remains your responsibility.

Firewall and Network Configuration

Coding agents require specific network access to function properly. If a coding agent is running behind a corporate firewall or on a restricted network, you may need to configure allowlists to enable coding agent functionality.

Built-in Agent Firewall

Coding agents run in a GitHub Actions environment with a built-in firewall that limits internet access by default. This firewall helps protect against:

- Data exfiltration
- Accidental leaks of sensitive information
- Execution of malicious instructions

By default, the agent's firewall allows access to:

- Common OS package repositories (Debian, Ubuntu, Red Hat, etc.)
- Popular container registries (Docker Hub, Azure Container Registry, AWS ECR, etc.)
- Language-specific package registries (npm, PyPI, Maven, RubyGems, etc.)
- Common certificate authorities for SSL validation

For the complete list of allowed hosts, see the Copilot allowlist reference.

Customizing Agent Firewall Settings

In your repository's "Coding agent" settings page, you can:

- Add custom hosts to the allowlist (for internal dependencies or additional registries)
- Opt out of the default recommended allowlist for stricter security
- Disable the firewall entirely (not recommended)

If a coding agent's request is blocked by the firewall, a warning will be added to the pull request or comment, detailing the blocked address and the command that triggered it.

For more information, see Customizing or disabling the firewall for GitHub Copilot coding agent.

Recommended URLs for Data Science Repositories

For data science and R-focused repositories, we recommend adding the following URLs to your Copilot allowlist. These sites are safe, reputable sources of documentation and packages that coding agents may need to access:

R Package Documentation and Ecosystems:

- `tidyverse.org` - `{tidyverse}` package documentation and learning resources
- `r-lib.org` - Core R infrastructure packages (`{devtools}`, `{testthat}`, `{usethis}`, etc.)
- `ggplot2.tidyverse.org` - `{ggplot2}` visualization package
- `dplyr.tidyverse.org` - `{dplyr}` data manipulation package
- `tidyr.tidyverse.org` - `{tidyr}` data tidying package
- `purrr.tidyverse.org` - `{purrr}` functional programming package
- `readr.tidyverse.org` - `{readr}` data reading package
- `stringr.tidyverse.org` - `{stringr}` string manipulation package
- `forcats.tidyverse.org` - `{forcats}` categorical data package

R Package Repositories:

- `cran.r-project.org` - The Comprehensive R Archive Network
- `cloud.r-project.org` - CRAN mirror (cloud-based)

- `docs.ropensci.org` - rOpenSci package documentation (e.g., `{targets}`)
- `rdatatable.gitlab.io` - `{data.table}` package documentation
- `rstudio.github.io` - RStudio-maintained packages (e.g., `{renv}`)

Code Style and Quality Tools:

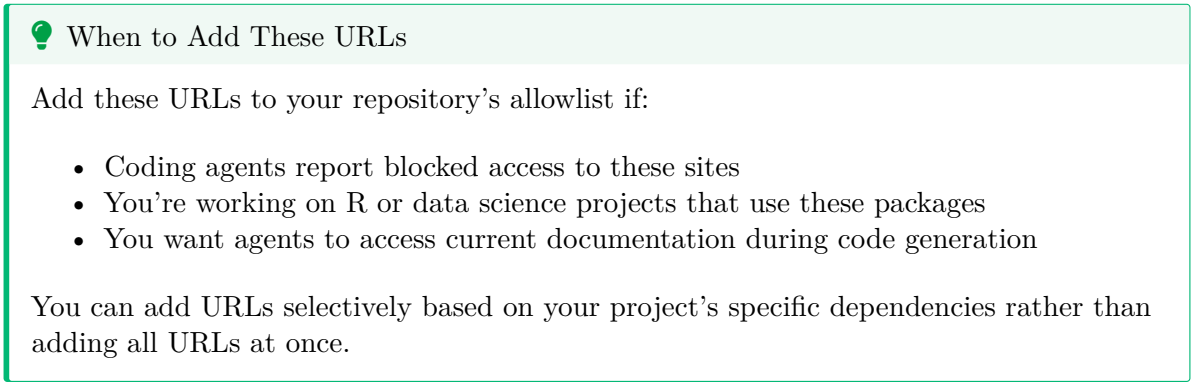
- `styler.r-lib.org` - `{styler}` code formatting package
- `lintr.r-lib.org` - `{lintr}` code linting package
- `roxygen2.r-lib.org` - `{roxygen2}` documentation package
- `style.tidyverse.org` - Tidyverse style guide

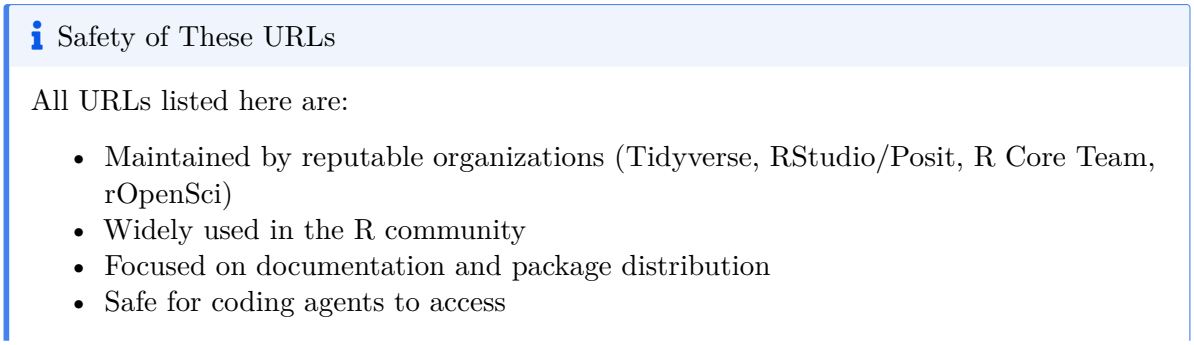
General Documentation and Reference:

- `en.wikipedia.org` - General reference and technical documentation
- `r-project.org` - Official R project website
- `quarto.org` - Quarto publishing system documentation
- `pandoc.org` - Pandoc document converter documentation

GitHub Organizations (for package repositories):

- `github.com/tidyverse/*` - Tidyverse package source code
- `github.com/r-lib/*` - R-lib package source code
- `github.com/rstudio/*` - RStudio package source code
- `github.com/ropensci/*` - rOpenSci package source code





These sites do not host user-generated content or allow arbitrary code execution, making them appropriate for inclusion in your allowlist.

Running Coding Agents Offline

Some environments restrict or prohibit internet access—high-performance computing (HPC) clusters, hospital networks, or air-gapped research servers may block connections to cloud AI providers. Running a local AI model lets you use coding assistance in these settings without sending code to external servers, which also addresses data-privacy concerns when working with sensitive or confidential data.

Trade-offs of Local vs. Cloud Models

Local models work best with a GPU (roughly 8 GB VRAM or more for smaller models); CPU-only inference is possible but significantly slower, and hardware needs vary with model size and quantization. They are generally less capable than frontier cloud models, and may produce lower-quality results on complex tasks. For routine work in fully connected environments, cloud-based agents remain the better choice. Use local models when network access or data-privacy policies require it.

Running a Local Model with Ollama

Ollama is a common way to run open-weight AI models locally. It packages models and a simple API server into a single tool and is available for macOS, Linux, and Windows.

Install Ollama:

Caution

Before running any remote install script, review it first. The real risk is piping `curl` straight into `sh`/`bash` (`curl ... | sh`), which executes unreviewed code. Save the script, read it, then run it: `curl -fsSL https://ollama.com/install.sh -o install.sh && less install.sh` (paging the saved file rather than piping `curl` into `less`, which can behave oddly in some terminal emulators). Alternatively, use your system package manager (e.g., `brew install ollama` on macOS) or follow the manual installation steps on the Ollama releases page.

```
# macOS / Linux: download, review, then run (do not chain these into one command)
curl -fsSL https://ollama.com/install.sh -o install.sh
```

```
less install.sh   # review the script before running it (see caution above)
sh install.sh
```

On Windows, download the installer from https://ollama.com/download.

Pull a code-focused model:

```
# Smaller, faster; works on most machines with a modern GPU or Apple Silicon
ollama pull qwen2.5-coder:7b

# More capable; larger memory footprint
ollama pull qwen2.5-coder:32b

# Alternatively, a general-purpose model (70B variant; needs a high-memory GPU)
ollama pull llama3.3
```

The VRAM each model needs depends on its size and quantization and changes as models are re-quantized—check the Ollama model library for current requirements. As a rough guide, smaller (7B) models run on consumer GPUs with around 8 GB of VRAM, while larger (32B and 70B) models need substantially more and may not fit on a single GPU.

Start the Ollama server:

```
ollama serve
```

By default the server listens at `http://localhost:11434`.

Connecting Positron Assistant to Ollama

Positron supports Ollama natively through the OpenAI-compatible API endpoint that Ollama exposes.

1. Open the Command Palette with `Cmd+Shift+P` (macOS) or `Ctrl+Shift+P` (Windows/Linux).
2. Run **Positron Assistant: Configure Language Model Providers**.
3. Select **Ollama** (or **Custom/OpenAI-compatible** if Ollama is not listed).
4. Set the base URL to `http://localhost:11434/v1` and leave the API key blank, or, if the client rejects an empty field, enter any placeholder value such as `ollama`.
5. Choose your model from the model list (e.g., `qwen2.5-coder:7b`).

Once configured, Positron Assistant will send requests to your local Ollama server instead of a cloud provider.

Connecting VS Code to Ollama via Continue

Continue is an open-source VS Code extension that supports Ollama and many other local and cloud backends.

1. Install the **Continue** extension from the VS Code marketplace.
2. Open the Continue sidebar and click the model selector.
3. Choose **Ollama** and select a model (Continue detects running Ollama models automatically).

Continue provides inline completions and a chat panel, similar to GitHub Copilot, but routed entirely to your local model.

Running an Autonomous Coding Agent with Aider

The editor integrations above provide inline completion and chat. For an autonomous agent that reads your files, proposes edits across a whole repository, and commits them to git—the local counterpart to a cloud coding agent—`aider` works directly against Ollama.

Install it in an isolated environment so its dependencies do not collide with other tools:

```
# Recommended: isolated install with pipx
pipx install aider-chat

# Or into your user environment with pip
python3 -m pip install --user aider-chat
```

Point it at Ollama and choose a model with the `ollama_chat/` prefix, which gives better results in `aider` than the plain `ollama/` prefix:

```
export OLLAMA_API_BASE=http://127.0.0.1:11434
aider --model ollama_chat/qwen2.5-coder:7b
```

! Raise the Ollama context window

By default Ollama uses a 2048-token context window, which silently truncates your code and makes the model look far less capable than it is. This is the single most common mistake when pairing `aider` with Ollama. Raise it with a model-settings file at `~/.aider.model.settings.yml`:

```
- name: ollama_chat/qwen2.5-coder:7b
  extra_params:
    num_ctx: 8192
```

> Larger values (16384, 32768) handle bigger files at the cost of more memory; pick the largest your machine can comfortably hold.

To avoid passing flags every time, set defaults in a config file at `~/.aider.conf.yml`:

```
model: ollama_chat/qwen2.5-coder:7b
set-env:
  - OLLAMA_API_BASE=http://127.0.0.1:11434
```

`aider` can also split the work between two models in "architect" mode: a larger model plans the change (the architect), and a second model applies the edits (the editor).

```
aider --architect \
  --model ollama_chat/qwen2.5-coder:32b \
  --editor-model ollama_chat/qwen2.5-coder:7b
```

This can improve results on multi-step changes, but on a machine without a strong GPU it roughly doubles the time per turn, because the two models take turns and their weights are swapped in and out of memory. Reserve it for genuinely tricky changes; for small edits, a single model is faster.

Falling Back Between Cloud and Local Automatically

Air-gapped work aside, the common case is a laptop that is usually online but sometimes is not—on a plane, behind a flaky hospital network, or temporarily rate-limited by a cloud provider. You can keep a coding agent working across these gaps by putting a cloud model and a local model behind one endpoint and falling back automatically.

LiteLLM runs a small local proxy that presents a single OpenAI-compatible endpoint. You give it a primary model and one or more fallbacks; when the primary fails with a retryable error—a rate-limit response (HTTP 429), or a connection error when you are offline—it retries the request on the next model. Pointing your agent at the proxy instead of directly at a provider makes the cloud-to-local switch automatic and invisible to the tool.

Install the proxy:

```
python3 -m pip install --user "litellm[proxy]"
```

Create a config file (for example, `~/.litellm/config.yaml`) with a cloud primary and a local fallback:

```yaml
model_list:
  # Cloud primary --- replace with your provider and a current model id
  - model_name: coder
    litellm_params:
      model: anthropic/YOUR-MODEL-ID
      api_key: os.environ/ANTHROPIC_API_KEY
  # Local fallback, served by Ollama
  - model_name: coder-local
    litellm_params:
      model: ollama_chat/qwen2.5-coder:7b
      api_base: http://localhost:11434

litellm_settings:
  num_retries: 2
  fallbacks:
    - coder: ["coder-local"]
```

Run the proxy, which listens on `http://localhost:4000`:

```
litellm --config ~/.litellm/config.yaml --port 4000
```

Then point your agent at the proxy. Because the proxy speaks the OpenAI API, most tools accept it as a custom endpoint. For `aider`:

```
export OPENAI_API_BASE=http://localhost:4000/v1
export OPENAI_API_KEY=placeholder   # the proxy needs no key unless you set one
aider --model openai/coder
```

Requests now go to the cloud model when it is reachable and fall back to the local model on a rate limit or when you are offline.

> **A cloud API key is separate from a chat subscription**
>
> The cloud side needs an API key billed per token, issued from the provider's developer console. A chat subscription such as Claude Pro or a Copilot seat is not an API key and cannot be used here. Omit the cloud entry entirely to run local-only, or add the key later to enable the hybrid.

Keep the proxy on loopback

By default the proxy binds to localhost, which is what you want. Do not expose it on `0.0.0.0` on a shared or untrusted network without authentication, because anyone who can reach the port can spend your cloud API key.

Working in HPC / Cluster Environments

Many HPC clusters do not have outbound internet access on compute nodes but do allow access on login nodes.

Install the Ollama binary on the cluster first

Ollama must be installed on the cluster (on the host that will run `ollama serve`) before any of the steps below. On most HPC systems you do not have root access, so the `curl | sh` installer may fail or install to the wrong place. Instead, check whether your cluster already provides it (e.g., `module load ollama`), ask your HPC administrators, or download a static binary from the Ollama releases page and place it on your `PATH`.

A useful pattern:

1. **Pre-pull models** on a login node or a machine with internet access, then copy the model files to the cluster:

```
# On a machine with internet access
ollama pull qwen2.5-coder:7b
# Ollama stores models in ~/.ollama/models by default
rsync -a ~/.ollama/models/ user@cluster.example.edu:~/.ollama/models/
```

Watch your home-directory quota

Model files are large—`qwen2.5-coder:7b` is ~4 GB and `qwen2.5-coder:32b` is ~20 GB—and most HPC home directories have tight quotas (often 10–50 GB). Filling your home directory can break other jobs. Redirect model storage to a scratch or project filesystem with `OLLAMA_MODELS` and rsync to that path instead:

```
# On your local machine: copy the models to the cluster scratch filesystem
rsync -a ~/.ollama/models/ user@cluster.example.edu:/scratch/$USER/ollama-models/
```

```
# On the cluster: point ollama at that path (export before `ollama serve`)
export OLLAMA_MODELS=/scratch/$USER/ollama-models
```

Set the same `OLLAMA_MODELS` value before running `ollama serve` so the server finds the models.

2. **Start Ollama on a compute node** (or an interactive session) using the pre-downloaded model files—no internet required. Set `OLLAMA_HOST=0.0.0.0` so the SSH tunnel from the login node can reach the port:

```
OLLAMA_HOST=0.0.0.0:11434 ollama serve
# If you redirected model storage (see quota warning above):
# OLLAMA_HOST=0.0.0.0:11434 OLLAMA_MODELS=/scratch/$USER/ollama-models ollama serve
```

   If you are on a shared compute node, be aware that binding to `0.0.0.0` exposes the Ollama port to other users on that host. Scheduler policies vary by site and job type, so confirm whether your job has exclusive node access (request it explicitly when in doubt—e.g., `--exclusive` in SLURM), or bind only to loopback (`OLLAMA_HOST=127.0.0.1:11434`) and tunnel from the login node when the node is shared.

3. **Forward the port** to your local machine to use your editor's Ollama integration. Because Ollama is running on a compute node (e.g., `gpu-node-01`), forward through the login node to that specific host:

```
# Replace gpu-node-01 with your actual compute node hostname
ssh -L 11434:gpu-node-01:11434 user@cluster.example.edu
```

   This terminal must stay open for as long as you use the editor's Ollama integration—closing it tears down the tunnel and silently drops the connection. Alternatively, start the tunnel in the background (non-interactive) so it does not occupy a terminal:

```
ssh -N -f -L 11434:gpu-node-01:11434 user@cluster.example.edu
```

   (`-N` runs no remote command, `-f` backgrounds ssh after authenticating.) To stop the tunnel later, match the full SSH command rather than a bare port string—`pkill -f "ssh.*-N.*11434:gpu-node-01"`—so you don't accidentally kill unrelated processes whose command line happens to contain that port. Safer still, note the PID when you start it (`pgrep -f "11434:gpu-node-01"`) and `kill` that PID directly.

   Then configure your editor to use `http://localhost:11434/v1` as the base URL.

> ⚠ SLURM and GPU Allocation
>
> If running Ollama on a SLURM-managed cluster, request a GPU node with enough VRAM for your chosen model and load any required CUDA modules before starting `ollama serve`. See the UCD-SERG Lab Manual's SLURM chapter for guidance on requesting GPU resources.

Privacy Considerations

Running a model locally ensures that your code and prompts never leave your machine or cluster. This is important when working with:

- Protected health information (PHI) or other HIPAA-regulated data
- Unpublished research data under data-use agreements (DUAs)
- Proprietary or commercially sensitive code

Even with local models, avoid including raw sensitive data in prompts. Work with anonymized or synthetic data wherever possible.

Configuring GitHub Copilot Settings

GitHub Copilot offers numerous configuration options that control how the AI assistant integrates into your development workflow. This section explains the key settings visible in your GitHub account preferences and provides guidance on which options to enable based on your use case.

Model Selection Options

GitHub Copilot provides access to multiple AI models, each with different capabilities and performance characteristics. The available models as of early 2026 include:

Anthropic Claude Models:

- **Claude Opus 4.1**: Most capable model for complex reasoning and analysis
  - *Pros*: Excellent at understanding nuanced requirements, handling complex codebases, superior code quality
  - *Cons*: Slower response times, may be overkill for simple tasks, limited availability (select option required)
  - *When to use*: Complex refactoring, architectural decisions, thorough code reviews
- **Claude Opus 4.5**: Latest version with enhanced capabilities
  - *Pros*: State-of-the-art performance, improved reasoning over 4.1
  - *Cons*: Similar trade-offs to Opus 4.1, requires selection
  - *When to use*: Most demanding tasks requiring cutting-edge capabilities
- **Claude Sonnet 4**: Balanced model optimizing capability and speed
  - *Pros*: Fast responses, strong performance, good default choice
  - *Cons*: Slightly less capable than Opus models for very complex tasks
  - *When to use*: General development work, most coding tasks
- **Claude Sonnet 4.5**: Enhanced version of Sonnet
  - *Pros*: Improved over Sonnet 4 while maintaining speed
  - *Cons*: Still not as powerful as Opus for extremely complex scenarios

  - *When to use*: Most daily development tasks

- **Claude Haiku 4.5**: Fast, efficient model for simpler tasks
  - *Pros*: Very fast responses, cost-effective, good for quick questions
  - *Cons*: Less capable for complex reasoning or large codebases
  - *When to use*: Simple completions, quick questions, repetitive tasks

OpenAI GPT Models:

As of 2026-05-08, OpenAI points users to ChatGPT Codex at `chatgpt.com/codex`. The OpenAI quickstart describes Codex as an AI coding assistant available through the Codex IDE extension that can read files, run commands, and write changes. This quickstart guide also links to a dedicated Codex app for working with local projects: OpenAI Codex quickstart guide. For additional background and platform context, Wikipedia describes Codex as an AI coding agent by OpenAI with desktop app availability on Windows and macOS as an additional access path: Codex (AI agent). In the GitHub Copilot model names shown below, the `-Codex` suffix identifies code-specialized variants (for example, `GPT-5.2-Codex` and `GPT-5-Codex`).

- **GPT-5.2-Codex**: Specialized for code generation
  - *Pros*: Strong code completion, good at common patterns
  - *Cons*: May hallucinate package names or APIs
  - *When to use*: Code completion, common coding patterns

- **GPT-5**: Latest general-purpose model
  - *Pros*: Broad knowledge, good general performance
  - *Cons*: Not specifically optimized for code
  - *When to use*: Mixed tasks involving code and documentation

- **GPT-5-Codex** (various versions including Mini and Max):
  - *Pros*: Specialized variants for different use cases
  - *Cons*: Fragmented options can be confusing
  - *When to use*: Specific scenarios where variant optimizations matter

Connecting Positron Assistant to OpenAI

If you are using Positron Assistant with OpenAI models, set up an OpenAI API key first.

Follow these steps:

1. Go to the OpenAI API keys page.
2. Sign in, choose or create the OpenAI project you want to use, and select **Create new secret key**.
3. Copy the key immediately and store it in a secure password manager.

4. In Positron, open the Command Palette with `Cmd+Shift+P` (or `Ctrl+Shift+P` on Windows/Linux).
5. Run `Positron Assistant: Configure Language Model Providers`.
6. Select **OpenAI**, paste your API key, and complete sign-in.

The Positron Assistant getting started guide states that OpenAI is enabled by default. If OpenAI does not appear as a provider, update Positron and confirm `positron.assistant.provider.openAI.enable` is not disabled.

Sources: Positron Assistant setup, OpenAI API key help, OpenAI quickstart.

Google Gemini Models:

- **Gemini 2.5 Pro**: High-capability model
  - *Pros*: Strong multimodal capabilities, good at understanding context
  - *Cons*: Less proven in coding scenarios than Claude or GPT
  - *When to use*: Tasks involving images or complex context
- **Gemini 3 Pro/Flash** (Preview): Latest generation
  - *Pros*: Cutting-edge capabilities, flash variant offers speed
  - *Cons*: Preview status means less stable, limited track record
  - *When to use*: Experimental workflows, evaluation of new capabilities

Lab Recommendation: For most lab work, enable **Claude Sonnet 4.5** as your default model. It provides excellent balance of capability and speed. Consider switching to **Claude Opus 4.5** for complex architectural decisions or difficult debugging sessions. Keep **Claude Haiku 4.5** enabled for quick inline completions.

Feature Settings

These settings control where and how Copilot integrates into your development environment:

Editor preview features:

- *What it does*: Enables previews of experimental features in your editor
- *Pros*: Access to latest capabilities before general release
- *Cons*: May have bugs or unstable behavior
- *Recommendation*: **Enable** if you're comfortable troubleshooting issues and want cutting-edge features

Copilot Chat in GitHub.com:

- *What it does*: Enables Copilot chat interface on GitHub.com
- *Pros*: Quick access to Copilot without opening an editor, useful for reviewing PRs
- *Cons*: Only available with paid license

- *Recommendation*: **Enable** (included in GitHub Copilot subscription)

Copilot CLI:

- *What it does*: GitHub Copilot for assistance in terminal
- *Pros*: AI help for command-line operations, shell commands, and git operations
- *Cons*: Requires separate installation and setup
- *Recommendation*: **Enable** and install via `gh extension install github/gh-copilot`

Copilot in GitHub Desktop:

- *What it does*: Enables Copilot in GitHub Desktop app
- *Pros*: AI assistance in GUI git client
- *Cons*: Limited compared to editor integration
- *Recommendation*: **Enable** if you use GitHub Desktop

Copilot Chat in the IDE:

- *What it does*: Enables chat interface in your code editor
- *Pros*: Context-aware help, refactoring assistance, code explanation
- *Cons*: Can be distracting if overused
- *Recommendation*: **Enable** (essential feature)

Copilot Chat in GitHub Mobile:

- *What it does*: Enables Copilot chat in mobile app
- *Pros*: Quick access on mobile devices
- *Cons*: Limited by mobile interface
- *Recommendation*: **Enable** for convenience

Copilot can search the web:

- *What it does*: Allows Copilot to search internet for up-to-date information
- *Pros*: Access to current documentation, recent library changes, latest best practices
- *Cons*: May introduce latency, results depend on search quality
- *Recommendation*: **Enable** for access to current information

Advanced Settings

Dashboard Entry Point:

- *What it does*: Allows instant chatting when landing on GitHub.com
- *Pros*: Quick access to Copilot without navigating menus
- *Cons*: None significant
- *Recommendation*: **Enable** for convenience

Copilot code review:

- *What it does*: Use Copilot to review your code and generate pull request summaries
- *Pros*: Automated code review suggestions, PR summary generation, catches common issues
- *Cons*: May generate false positives, shouldn't replace human review
- *Recommendation*: **Enable** (major productivity boost)

Automatic Copilot code review:

- *What it does*: Automatically reviews all pull requests you create
- *Pros*: Catches issues early without manual triggering
- *Cons*: May be noisy on simple PRs, uses API quota
- *Recommendation*: **Disable** initially; enable only after you're comfortable with code review quality

Copilot coding agent:

- *What it does*: Delegate tasks to Copilot coding agent in repositories where it is enabled
- *Pros*: Autonomous multi-file edits, can execute complex refactoring, runs tests and fixes issues
- *Cons*: Requires careful oversight, can make unwanted changes if instructions unclear
- *Recommendation*: **Enable** (see Section for safe usage guidelines)

Copilot Memory (Preview):

- *What it does*: Remember repository context across Copilot agent interactions
- *Pros*: Better context awareness, learns repository patterns and conventions
- *Cons*: Preview feature governed by pre-release terms, potential privacy implications
- *Recommendation*: **Enable** to help Copilot learn your codebase patterns

MCP servers in Copilot:

- *What it does*: Connect MCP servers to Copilot in all editors and Coding Agent
- *Pros*: Extend Copilot with custom tools and integrations
- *Cons*: Requires MCP server setup and maintenance
- *Recommendation*: **Enable** if you have MCP servers configured; otherwise this setting has no effect

Copilot-generated commit messages:

- *What it does*: Allow Copilot to suggest commit messages when you make changes
- *Pros*: Saves time, generates descriptive messages based on code changes
- *Cons*: May miss important context, still requires review
- *Recommendation*: **Enable** but always review and edit suggested messages

Copilot Spaces:

- *What it does*: View and create Copilot Spaces (collaborative AI environments)
- *Pros*: Share AI context with team members
- *Cons*: Additional complexity for individual work
- *Recommendation*: **Enable** for team collaboration features

Copilot Spaces Individual Access:

- *What it does*: Create individually owned Copilot Spaces
- *Pros*: Personal AI workspaces for complex projects
- *Cons*: May fragment your workflow
- *Recommendation*: **Enable** for flexibility

Copilot Spaces Individual Sharing:

- *What it does*: Share individually owned Copilot Spaces
- *Pros*: Collaborate while maintaining ownership
- *Cons*: None significant
- *Recommendation*: **Enable** for sharing capability

Summary of Recommended Settings

For lab members, we recommend the following configuration:

Enable these features:

- All Copilot Chat options (GitHub.com, CLI, IDE, Mobile)
- Web search capability
- Dashboard Entry Point
- Copilot code review (but not automatic review initially)
- Copilot coding agent
- Copilot Memory
- MCP servers (if configured)
- Copilot-generated commit messages
- All Copilot Spaces options

Model selection:

- Default: Claude Sonnet 4.5
- Complex tasks: Claude Opus 4.5
- Quick completions: Claude Haiku 4.5

Enable with caution:

- Editor preview features (only if comfortable with potential instability)

- Automatic Copilot code review (wait until familiar with review quality)

Following these guidelines will help establish an effective Copilot configuration. The key is to enable features that add value to your workflow while maintaining awareness that AI assistance requires validation (see Section).

Connecting VS Code to a Custom Model Endpoint (BYOK)

VS Code's built-in Chat usually talks to GitHub's hosted models. It can also route requests to a model provider of your own; GitHub calls this "bring your own key" (BYOK). The lab uses BYOK to reach Databricks model serving endpoints, which expose an OpenAI-compatible API, through the community extension `oai-compatible-copilot`.

This section describes the wiring and three errors you are likely to hit.

Wiring the extension to Databricks

Databricks serves models over an OpenAI-compatible endpoint at `https://<workspace>.cloud.databricks.com/serving-endpoints`. Point the extension at it in VS Code `settings.json`:

```
"oaicopilot.baseUrl": "https://<workspace>.cloud.databricks.com/serving-endpoints",
"oaicopilot.models": [
  {
    "id": "databricks-claude-sonnet-4-6",
    "owned_by": "databricks",
    "apiMode": "openai"
  }
]
```

The `id` of each model must exactly match the name of a deployed serving endpoint in your workspace. The extension sends `id` as the OpenAI `model` field, and Databricks routes `POST /serving-endpoints/chat/completions` to the endpoint of that name. An `id` that names no real endpoint fails (see the 404 below).

To store your token, run **Set OAI Compatible Multi-Provider Apikey** from the Command Palette (`Cmd+Shift+P` / `Ctrl+Shift+P`), choose the `databricks` provider, and paste a Databricks personal access token. The extension keeps it in VS Code's encrypted secret storage (under `oaicopilot.apiKey.databricks`) and sends it as an `Authorization: Bearer` header.

Three errors, and why they stack

These failures sit on top of each other: fixing one uncovers the next, so work through them top-down.

1. "No utility model is configured for 'copilot-utility-small'"

When your main Chat model is a BYOK model, VS Code still needs a small "utility" model for background chores such as generating the conversation title and naming git branches. If none is configured, Chat fails before it ever reaches your provider:

```
No utility model is configured for 'copilot-utility-small'
while the selected main agent model is BYOK.
```

Set `chat.byokUtilityModelDefault` in `settings.json`:

- `"mainAgent"`: reuse your BYOK main model for these chores. This keeps all traffic on your provider and needs no extra endpoint, so it is the simplest choice.
- `"copilot"`: use GitHub's hosted utility model. This needs an active Copilot subscription.
- `"none"`: the default, which errors on purpose.

This requirement arrived in a mid-2026 VS Code update. Before that, BYOK chat worked without the setting, so an editor update can make a working setup start failing here.

2. `[404] ENDPOINT_NOT_FOUND`

```
[404] Not Found
{"error_code":"ENDPOINT_NOT_FOUND",
 "message":"The given endpoint does not exist, please retry after
            checking the specified model and version deployment exists."}
```

The `model` name in the request is not a serving endpoint that exists in the workspace. Check that every `id` in `oaicopilot.models` matches a real, deployed endpoint (Databricks workspace → **Serving**), and remove or rename any entry that points at a name with no deployment. A stray placeholder entry, such as a leftover `copilot-utility-small`, is a common cause.

3. `[403] Invalid access token`

```
[403] Forbidden
{"error_code":403,"message":"Invalid access token."}
```

The stored token is expired or revoked. Databricks OAuth tokens are short-lived and can expire within the day, so a session that worked in the morning can start returning 403 by afternoon; personal access tokens last until their configured expiry. Generate a fresh token (Databricks → **Settings** → **Developer** → **Access tokens**) and re-run **Set OAI Compatible Multi-Provider Apikey**. Prefer a long-lived personal access token to avoid frequent re-authentication. No window reload is needed; the extension reads the token on each request.

> **Tip**
>
> A quick way to tell 404 from 403: a 404 means the request authenticated but named a missing endpoint (a model-name or configuration problem), while a 403 means the token itself was rejected (an authentication problem).

Configuring the Agent Environment

The `.github/workflows/copilot-setup-steps.yml` file allows you to customize the development environment in which the GitHub Copilot coding agent operates. This file preinstalls tools and dependencies so that Copilot can build, test, and lint your code more reliably.

Why Configure the Environment?

While Copilot can discover and install dependencies through trial and error, this can be slow and unreliable. Additionally, Copilot may be unable to access private dependencies. Preconfiguring the environment ensures:

- Faster agent startup and execution
- More reliable builds and tests
- Access to private or authenticated dependencies
- Consistent development environment across all agent sessions

File Location and Structure

The workflow file must be located at `.github/workflows/copilot-setup-steps.yml` in your repository's **default branch**. It follows GitHub Actions workflow syntax but must contain a single job named `copilot-setup-steps`.

Basic Configuration Example

See this repository's own `.github/workflows/copilot-setup-steps.yml` for a configuration adapted for R and Quarto projects.

Using `actions/checkout`

The `actions/checkout` action is used to check out your repository code so that the workflow can access it. While Copilot will automatically check out your repository if you don't include this step, **explicitly including it is necessary** when your setup steps need to access repository files.

Why explicitly include checkout?

Many dependency installation steps require access to repository files:

- `r-lib/actions/setup-renv@v2` needs `renv.lock` to install R package dependencies
- `r-lib/actions/setup-r-dependencies@v2` needs `DESCRIPTION` to install R package dependencies
- `npm ci` needs `package-lock.json` to install Node.js dependencies
- `pip install -r requirements.txt` needs the requirements file

Without an explicit checkout step, these dependency installation commands will fail because the necessary files won't be available yet.

Basic checkout:

```yaml
- name: Checkout code
  uses: actions/checkout@v4
```

Important: The Copilot coding agent overrides any `fetch-depth` value you set in the checkout step. According to GitHub's official documentation, this override happens "to allow the agent to rollback commits upon request, while mitigating security risks." The agent dynamically determines the appropriate fetch depth based on the pull request context.

While you cannot control the fetch depth used by Copilot, the agent still has access to sufficient git history to perform its work effectively, including comparing changes and understanding the context of your pull request.

Configurable Options

You can customize only these specific settings in the `copilot-setup-steps` job:

- `steps`: Setup commands and actions to run
- `permissions`: Access permissions (typically `contents: read`)
- `runs-on`: Runner type (Ubuntu x64 Linux only)
- `services`: Database or service containers
- `snapshot`: Save environment state
- `timeout-minutes`: Maximum 59 minutes

All other workflow settings are ignored by Copilot.

Common Setup Tasks

For Node.js/TypeScript projects:

```yaml
- name: Set up Node.js
  uses: actions/setup-node@v4
  with:
    node-version: "20"
    cache: "npm"

- name: Install dependencies
  run: npm ci
```

For Python projects:

```yaml
- name: Set up Python
  uses: actions/setup-python@v5
  with:
    python-version: "3.11"

- name: Install dependencies
  run: pip install -r requirements.txt
```

For R projects:

```yaml
- name: Set up R
  uses: r-lib/actions/setup-r@v2
  with:
    r-version: 'release'

- name: Install R dependencies
  uses: r-lib/actions/setup-renv@v2
```

Environment Variables and Secrets

To set environment variables for Copilot:

1. Navigate to your repository's **Settings**
2. Go to **Environments**
3. Select or create the `copilot` environment
4. Add environment variables or secrets as needed

Use secrets for sensitive values like API keys or passwords.

Testing Your Configuration

The workflow runs automatically when you modify `copilot-setup-steps.yml`, allowing you to validate changes in pull requests. You can also manually trigger the workflow from the repository's **Actions** tab.

Setup logs appear in the agent session logs when Copilot starts working. If a step fails, Copilot will skip remaining steps and begin working with the current environment state.

Advanced Configuration

Larger runners: For projects requiring more resources, you can use larger GitHub-hosted runners:

```yaml
jobs:
  copilot-setup-steps:
    runs-on: ubuntu-4-core
```

Self-hosted runners (ARC): For access to internal resources or private registries, use Actions Runner Controller (ARC) self-hosted runners:

```yaml
jobs:
  copilot-setup-steps:
    runs-on: arc-scale-set-name
```

Note: When using self-hosted runners, you must disable Copilot's integrated firewall in repository settings and configure appropriate network security controls.

Git Large File Storage (LFS): If your repository uses Git LFS:

```yaml
- uses: actions/checkout@v4
  with:
    lfs: true
```

Further Reading

For complete details, see Customizing the development environment for GitHub Copilot coding agent.

Agent Skills

Agent Skills are a lightweight, open standard for extending AI agent capabilities with specialized knowledge and workflows. The specification defines a portable, tool-agnostic format that any compatible agent can load.

What is a Skill?

At its core, a skill is a folder containing a `SKILL.md` file. This file includes the `name` and `description` metadata required by the specification, along with instructions that tell an agent how to perform a specific task. Skills can also bundle supporting resources:

- Scripts
- Reference documentation
- Templates and assets

```
my-skill/
   SKILL.md          # Required: metadata + instructions
   scripts/          # Optional: executable code
   references/       # Optional: documentation
   assets/           # Optional: templates, resources
```

How Agents Load Skills

The specification describes a *progressive disclosure* model, in which an agent typically loads a skill in three stages:

1. **Discovery**: At startup, the agent loads only the name and description of each available skill — just enough to know when it might be relevant.
2. **Activation**: When a task matches a skill's description, the agent reads the full `SKILL.md` instructions into context.
3. **Execution**: The agent follows the instructions, optionally executing bundled scripts or loading referenced files as needed.

This means full instructions load only when needed, so agents can maintain many skills with a small context footprint.

Why Use Skills?

Skills package procedural knowledge and team-specific context into portable, version-controlled folders. This gives agents:

- **Domain expertise**: Capture specialized knowledge as reusable instructions
- **Repeatable workflows**: Turn multi-step tasks into consistent, auditable procedures
- **Cross-tool reuse**: Build a skill once and use it across any skills-compatible agent

Further Reading

For the complete specification and more details, see agentskills.io.

The d-morrison/ai-config repository contains an example of personal Claude Code configuration, including user-level skills and slash commands, synced across machines via Git.

Claude Code Cloud Environments

Claude Code is a CLI coding agent that can also run tasks on Anthropic-managed cloud infrastructure— either from the web at claude.ai/code ("Claude Code on the web"), or from the terminal by adding the `--remote` flag to move a session into the cloud.

Each cloud run executes inside a configured **environment**. An environment bundles three things:

- **Network access level**: what the cloud session is allowed to reach (a security control, analogous to the firewall configuration described above).
- **Environment variables**: values the session needs at runtime, such as `NODE_ENV`, database URLs, or API keys.
- **Setup scripts**: Bash that runs automatically when the session starts, for example to install dependencies.

Selecting the default environment with `/remote-env`

The `/remote-env` slash command sets **which configured environment is the default** for `--remote` runs:

- With a single environment configured, it shows your current configuration.
- With multiple environments, it opens an interactive picker so you can choose the default.

`/remote-env` only *selects* the default; to add, edit, or archive the environments themselves, use the web interface at claude.ai/code. Because `/remote-env` opens an interactive panel, run it from an interactive `claude` terminal session.

> **Availability**
>
> Claude Code on the web (and the cloud environments it relies on) is a research-preview feature, available to Pro, Max, and Team users (and Enterprise users with eligible seats). Availability and behavior may change.

For details, see the Claude Code on the web documentation and the slash command reference.

When to use a coding agent

Coding agent sessions are currently[1] considered "premium requests", which are limited resources; see https://github.com/features/copilot/plans for details. So, use coding agents sparingly. Use them for complex changes that would be difficult or time-consuming for you to complete by hand. Coding agents also take time to get configured for work, every time you make a request. See https://docs.github.com/en/copilot/how-tos/use-copilot-agents/coding-agent/customize-the-agent-environment#preinstalling-tools-or-dependencies-in-copilots-environment for ways to reduce that startup time, but it will never be 0. If you can complete the task faster than the coding agent can, you should probably do it yourself. For example, when you have errors in the spell-check or lint workflows, you can often fix them faster than Copilot can. Similarly, when reviewing Copilot's PRs, you can often make direct changes to the branch faster than you could write clear review comments and get Copilot to address them.

Also, the less we practice, the weaker our skills get, and the harder it is for us to supervise the agents and make sure they are actually doing what we want them to do, the way we want them to do it. You should exercise your own coding skills regularly, just like you would for any other skill you want to maintain.

Editing with `.docx` files

GitHub Copilot coding agents can read Microsoft Word (`.docx`) files, including tracked changes and comments. This enables a hybrid editing workflow where:

1. Lab members can export Quarto content to Word format for review
2. Reviewers can make edits, add tracked changes, and insert comments in Word
3. Coding agents can read the `.docx` file and translate the edits back to Quarto format

When using this workflow, make sure to explicitly instruct the coding agent to:

- Examine and apply all tracked changes in the `.docx` file
- Read and address all comments in the `.docx` file

[1] 2026-01-10

- Translate edits from Word formatting to appropriate Quarto/markdown syntax

This approach makes it easier for collaborators who are more comfortable with Word to contribute while maintaining the source files in Quarto format.

Known Issue: "Document 1" Warning in Word

When opening DOCX files generated by Quarto (including this site), Microsoft Word may display a warning message and open the file with the title "Document 1" instead of the actual filename. Word may also require you to save the file before you can add comments or track changes.

This is a known limitation with how Quarto generates DOCX files. The issue is being tracked in the Quarto project:

- Quarto CLI Issue #6357
- Quarto Discussion #6544
- Quarto CLI Issue #10587

Workaround: If you are the author generating the DOCX file from Quarto, follow these steps before sharing with collaborators:

1. Open the generated DOCX file in Microsoft Word
2. Immediately save the file (File → Save, or Ctrl+S/Cmd+S)
3. Close and re-open the file to verify it no longer shows "Document 1"
4. Share this saved version with collaborators

This one-time step ensures that when collaborators open the file, they won't see the "Document 1" warning and can immediately add comments and track changes without issues.

Copilot Instructions for this Repository

A `.github/copilot-instructions.md` file contains repository-specific instructions and guidelines for GitHub Copilot coding agents. This file helps ensure that AI-generated contributions follow the project's formatting standards, coding conventions, and documentation practices.

For a Quarto-based repository like this one, the copilot instructions file typically specifies:

- Markdown and Quarto formatting rules (e.g., blank lines before lists, line breaks in prose)
- R code style guidelines (e.g., using native pipe `|>`, following tidyverse style)
- File organization patterns (e.g., using Quarto includes for modular content)
- How to work with DOCX files for hybrid editing workflows
- Version control best practices (e.g., do not commit rendered artifacts that are generated only for preview/review)

- Repository-specific best practices

By having these instructions in `.github/copilot-instructions.md`, you ensure that coding agents produce consistent, high-quality contributions that align with the project's established practices. This reduces the review burden and helps maintain consistency across all contributions, whether made by humans or AI assistants.

See this repository's own `.github/copilot-instructions.md` for a working example.

Using Copilot Review Before Human Review

Before requesting review from other humans, **always have Copilot review your pull request first**—even if Copilot created the PR itself. AI review provides fast, thorough feedback that helps catch issues before involving human reviewers, saving everyone time and improving code quality.

Why review with Copilot first:

- **AI has more bandwidth**: Copilot can review code immediately without competing priorities
- **Catch common issues early**: Copilot excels at identifying bugs, logic errors, security vulnerabilities, and style inconsistencies
- **Improve human review quality**: When humans review cleaner code, they can focus on higher-level concerns like design and architecture rather than basic issues
- **Learn from feedback**: Even experienced developers benefit from Copilot's perspective on best practices and potential improvements
- **Growing capabilities**: AI review capabilities continue to improve over time, making this investment increasingly valuable

Copilot review workflow:

1. **Assign Copilot as a reviewer**: On your pull request page, assign Copilot to review the PR the same way you would assign any other reviewer. Click "Reviewers" in the right sidebar and select Copilot from the list.
2. **Review Copilot's comments**: Once Copilot completes its review, carefully examine each comment. For each comment, decide whether you agree with the suggestion:
    - **If the comment is correct**: Address it by making code changes yourself or ask Copilot to apply the fix using GitHub's suggestion features
    - **If the comment is incorrect or not applicable**: Dismiss the comment with an explanation for why it doesn't apply
    - **If you're uncertain**: Seek a second opinion from a human reviewer or do additional research

3. **Request another Copilot review**: After addressing or dismissing all comments, request another review from Copilot. This creates an iterative improvement process.
4. **Iterate until satisfied**: Repeat the review-and-address cycle until Copilot stops providing valuable suggestions. This typically takes 1-3 iterations depending on the complexity of the changes.
5. **Request human review**: Only after you've addressed Copilot's feedback should you request review from human team members. At this point, the code should be in better shape, allowing human reviewers to focus on higher-level concerns.

Important considerations:

- **Copilot isn't perfect**: AI review can produce false positives or miss important issues. Always apply your own judgment when evaluating Copilot's suggestions.
- **Don't blindly accept all suggestions**: Some of Copilot's recommendations may not fit your specific context or requirements. It's perfectly appropriate to dismiss comments that don't apply.
- **Human review remains essential**: Copilot review supplements but does not replace human code review. Humans bring domain knowledge, understanding of business requirements, and judgment about trade-offs that AI cannot replicate.
- **Document dismissals**: When dismissing Copilot comments, briefly explain why. This helps human reviewers understand your reasoning and can serve as documentation for future reference.

For pull request authors:

Even if you're highly experienced, treating Copilot review as a required pre-review step helps maintain code quality and makes the best use of everyone's time. The few minutes spent on Copilot review often save hours of back-and-forth with human reviewers.

For human reviewers:

When you receive a PR for review, check whether the author has completed the Copilot review process. If Copilot hasn't reviewed the PR yet, consider asking the author to complete that step first before you invest time in review. This ensures you're reviewing code that has already been through initial automated quality checks.

Reviewing a Copilot PR You Didn't Create

When reviewing a pull request where someone else prompted Copilot to make changes, follow these guidelines to avoid confusion and ensure smooth collaboration:

Understanding PR roles:

The general PR roles (issue creator, author, reviewer, merger, assignee, and PR steward) are described in the UCD-SERG Lab Manual's GitHub chapter. Copilot-assisted workflows add two more:

- **PR prompter**: Assigns a developer (human or AI) to start working on a PR, often by assigning an issue to Copilot. The PR prompter is sometimes the same person as the issue creator, but is often a project maintainer who reviews and triages external issue reports. When Copilot creates commits, both Copilot and the prompter are listed as co-authors. Typically the PR prompter becomes the PR manager.
- **PR manager**: Supervises and guides the PR authors, assigns reviewers, and controls the PR workflow, deciding when and how Copilot makes additional changes. The PR manager can hand off this role to someone else.

The scenario:

- One team member (the "PR prompter") assigned Copilot to work on an issue or explicitly prompted Copilot to start working
- The prompter may or may not be the same person who originally created the issue
  - In projects with a user base, users often submit issues (bug reports, feature requests)
  - A project maintainer then steps in, adds their perspective, and assigns the issue to Copilot
  - In this case, the maintainer who assigned Copilot is the prompter, not the original issue creator
- Copilot created the PR with the prompter as co-author
- The prompter (now acting as PR manager) requested your review
- Copilot may have also automatically reviewed the PR

As a non-manager reviewer, your role is to provide feedback, not to directly initiate more work by Copilot. The PR manager should remain in control of when and how Copilot makes additional changes.

Recommended review workflow:

1. **Use "Comment" or "Request changes" based on the severity of issues**:
   - Use **"Comment"** for suggestions, questions, or minor issues that don't block merging
   - Use **"Request changes"** for significant issues that must be addressed before merging
   - Both options allow you to provide feedback without directly triggering Copilot
2. **Don't ask Copilot to make changes directly**:
   - Avoid using features that would trigger Copilot to start working immediately

- Let the PR manager decide whether to ask Copilot to address your comments or make changes themselves

3. **Write clear, actionable comments**:
    - Explain what needs to change and why
    - Suggest specific solutions when appropriate
    - The PR manager will decide how to address your feedback

For PR managers:

After receiving reviews from other team members:

1. **Review all comments carefully**:
    - Decide which comments you agree with
    - Dismiss or respond to comments you don't entirely agree with
    - This ensures Copilot only addresses feedback you've validated
2. **Choose how to address valid feedback**:
    - **Option A**: Make the changes yourself (faster for simple fixes)
    - **Option B**: Ask Copilot to address the feedback (better for complex changes)
    - **Option C**: Add your own review summarizing which comments Copilot should address, then ask Copilot to respond to the open comment threads
3. **Maintain clear communication**:
    - Let reviewers know how you plan to address their feedback
    - Mark conversations as resolved after addressing them
    - Request re-review from humans after Copilot makes significant changes
    - Update the PR's "Assignees" field to reflect who is currently responsible for the PR

Transferring the PR manager role:

The original PR manager can hand over a PR to another person, who then becomes the new PR manager with control over Copilot's work on that PR. This might be useful when:

- The original PR manager is unavailable or on leave
- Someone with different expertise needs to guide the remaining work
- Responsibilities are being redistributed within the team

To transfer the PR manager role:

1. The original PR manager should clearly communicate the handover to all reviewers
2. The new PR manager should review the PR's history and any open feedback
3. The new PR manager should take over responding to Copilot and managing future iterations

4. The team should update the PR's "Assignees" field and comments or description to reflect the current PR manager

This workflow ensures the PR manager maintains control over the development process while benefiting from collaborative human review and Copilot's implementation capabilities.

Installing Claude Code on Windows

Claude Code is Anthropic's command-line coding agent. Installing it on Windows works well, but a few platform-specific pitfalls can cost you hours if you don't know about them. These notes capture a setup that works, and the gotchas to watch for.

> **Note**
>
> These notes were written in June 2026. As with the rest of this chapter, treat the specifics with caution— installers and behavior change quickly.

Run it from a Unix-like terminal

Claude Code expects a Unix-like shell. On Windows, run it from one of:

- **Git Bash** (ships with Git for Windows);
- **MSYS2** (https://www.msys2.org/), which also gives you a package manager (`pacman`) and optional `zsh`;
- **WSL** (Windows Subsystem for Linux), the most Unix-faithful option.

If you use WSL, install Claude Code *inside* WSL— a Windows install will not carry over, because WSL has its own filesystem and `PATH`. Inside WSL the standard Linux install applies, and the Windows-specific `PATH` and `rehash` gotchas below don't apply.

Install Claude Code

There are two common routes:

1. **Native installer** (recommended):

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

   This installs the binary to `~/.local/bin`. If your organization's policy requires it, download and inspect the script before running it rather than piping it straight to `bash`.

2. **npm** (requires Node.js):

```
npm install -g @anthropic-ai/claude-code
```

> **!** Claude Code migrates itself out of npm
>
> Even if you install via npm, current versions **migrate themselves to a native install** the first time you run `claude` (at `~/.local/bin/claude`, or `~/.local/bin/claude.exe` on Windows) and remove the npm copy. You can watch this happen in the terminal output the first time you run `claude`; the current install methods are documented in the Claude Code setup guide.
> This is the single most confusing Windows gotcha: a path that worked a moment ago "disappears." **Do not** hardcode the npm location (e.g. `.../AppData/Roaming/npm/...`) in your shell config. Point your `PATH` at the shell-appropriate directory shown in the next section instead.

Make sure your shell can find `claude`

The native binary lives in `~/.local/bin`. The installer adds this directory to `PATH` for ordinary shells, but on Windows two things commonly break that.

MSYS2 does not inherit the Windows `PATH` by default. It starts in a "minimal" path mode, so tools installed elsewhere on Windows are invisible to it. Add the binary's directory explicitly in your `~/.zshrc` (or `~/.bashrc`). Because MSYS2's `$HOME` is `/home/<you>` (its own home, not the Windows profile where the installer puts the binary), point `PATH` at the absolute Windows path:

```
export PATH="/c/Users/<you>/.local/bin:$PATH"     # MSYS2
```

In **Git Bash**, `$HOME` already is `/c/Users/<you>` (your Windows user profile), so the shorthand works there:

```
export PATH="$HOME/.local/bin:$PATH"     # Git Bash
```

Rehash after changing `PATH`. `zsh` and `bash` cache the locations of executables. If you add a directory to `PATH` in your shell config, the shell may still report `command not found` *even though the directory is on `PATH`*, because its command table is stale. Force a rebuild in the same shell right after editing `PATH`:

```
rehash        # zsh; use 'hash -r' in bash
```

This bites hardest with oh-my-zsh, which builds zsh's command table *before* your custom `PATH` line runs, so opening a fresh window may not clear the stale `command not found` on its own until you `rehash` (or move the `PATH` line before oh-my-zsh initializes).

Do not edit dotfiles with PowerShell redirection

> ⚠ PowerShell writes the wrong encoding
>
> Never write to `~/.bashrc`, `~/.zshrc`, or other shell config files using PowerShell's `>` or `>>` redirection. Windows PowerShell writes **UTF-16 with a byte-order mark**, which `bash`/`zsh` read as a stray character at the start of the file:
>
> `bash: $'\377\376export': command not found`
>
> Edit dotfiles from *inside* the shell (e.g. with `nano`, `vim`, or `echo 'export PATH=...' >> ~/.zshrc` run in bash/zsh), or with an editor that saves UTF-8 without a BOM (e.g. VS Code).

Authenticating the GitHub CLI in Git Bash or MSYS2

If you also use the GitHub CLI (`gh`) to push code or open pull requests, `gh auth login` may fail with:

`could not prompt: … running in MinTTY without pseudo terminal support`

Git Bash and MSYS2 both default to the MinTTY terminal, which can't host the interactive prompt. Wrap the command with `winpty`, or run it from PowerShell / Windows Terminal instead:

```
winpty gh auth login
```

`winpty` ships with Git Bash, but in MSYS2 it's a separate package — install it first with `pacman -S winpty` if you get `command not found`.

Verify the install

Open a **new** terminal window (so it picks up your updated config) and run:

```
claude --version      # prints the installed version number
```

If you get a version number, you're ready to run `claude` in your project directory. If you get `command not found`, re-check the two `PATH` issues above: the directory must be on `PATH`, and you must `rehash` (or open a fresh window) after changing it.

References

2001: A Space Odyssey. 1968. Film. https://en.wikipedia.org/wiki/2001:_A_Space_Odyssey_(film).

Asimov, Isaac. 1950. *I, Robot*. Novel; Gnome Press. https://search.library.ucdavis.edu/permalink/01UCD_INST/9fle3i/alma990000226350403126.

Battlestar Galactica. 2004. Television Series. https://en.wikipedia.org/wiki/Battlestar_Galactica_(2004_TV_series).

Blade Runner. 1982. Film. https://en.wikipedia.org/wiki/Blade_Runner.

Card, Orson Scott. 1985. *Ender's Game*. Novel; Tor Books. https://en.wikipedia.org/wiki/Ender%27s_Game.

Flight of the Conchords. 2007. *The Humans Are Dead*. Music Video. https://www.youtube.com/watch?v=B1BdQcJ2ZYY.

Herbert, Frank. 1965. *Dune*. Novel; Chilton Books. https://en.wikipedia.org/wiki/Organizations_of_the_Dune_universe#Thinking_machines.

LeCun, Yann. 2022. *A Path Towards Autonomous Machine Intelligence*. Meta AI Research; New York University; Technical Report. https://openreview.net/forum?id=BZ5a1r-kVsf.

Terminator 3: Rise of the Machines. 2003. Film. https://en.wikipedia.org/wiki/Terminator_3:_Rise_of_the_Machines.

The Matrix. 1999. Film. https://en.wikipedia.org/wiki/The_Matrix.

WarGames. 1983. Film. https://en.wikipedia.org/wiki/WarGames.