# Responsible Use of Generative AI in Assisted Coding

Generative AI tools are transforming how researchers write code. From simple chatbot interactions to fully autonomous coding agents, these tools offer powerful capabilities … and with great power comes great risks and responsibilities.

This lesson provides a practical framework for researchers who want to use AI coding assistants in their day-to-day software work with research. The framwork stresses on the importance of maintaining control, security, and research integrity. We will explore three scenarios of increasing automation and decreasing user control, helping you make informed decisions about which approach fits your needs.

```
+------------------+    +------------------+    +------------------+
|  Scenario I      |    |  Scenario II     |    |  Scenario III    |
|  Full Control    |    |  IDE Integration |    |  Agentic AI      |
|                  |    |                  |    |                  |
|  Chat + manual   |    |  Tab completion  |    |  Autonomous      |
|  copy/paste      |    |  inline suggest  |    |  code agents     |
|                  |    |                  |    |                  |
|  LOW RISK        |    |  MEDIUM RISK     |    |  HIGH RISK       |
|  HIGH CONTROL    |    |  MEDIUM CONTROL  |    |  LOW CONTROL     |
+------------------+    +------------------+    +------------------+
```

*ASCII created with Claude. It should be replaced with an image*

**⚙ Prerequisites**

- Basic familiarity with programming (Python examples used, replace with your favourite language)
- Access to a code editor (e.g. VS Code) or Jupyter environment
- Optional: Accounts for AI tools you want to try (ChatGPT, Claude, GitHub Copilot, etc. Please note that some tools are not free and require a credit card).

**Warning**

This is work in progress. Known limitations

1. There might be too much content than what can be covered during the CodeRefinery workshop.
2. Some tools change so fast, this is likely already obsolete in two months from now.

3. Many bits here and there need testing and improvement.

| | |
|---|---|
| xx min | Introduction to LLMs for Code |
| xx min | Scenario I: Full Control (Chat-Based Coding) |
| xx min | Scenario II: IDE Integration (Tab Completion and Inline Suggestions) |
| xx min | Scenario III: Full Agentic Code Development |
| xx min | security |
| xx min | conclusion |

## Introduction to LLMs for Code

❓ **Questions**

- What are Large Language Models (LLMs) and how do they generate code?
- How are coding-specific models trained and fine-tuned?
- What tools are available for AI-assisted coding?

❗ **Objectives**

- Understand the basic architecture and training process of code LLMs
- Learn about open datasets used to train coding models
- Get an overview of the current landscape of AI coding tools
- Recognize the fundamental limitations and capabilities of these models

### What is a Large Language Model?

A Large Language Model (LLM) is a type of artificial intelligence trained on vast amounts of text data to predict and generate human-like text. At their core, these models learn statistical patterns in language: given a sequence of words (or better *tokens*: fragments of words, commas, and anything in text), they predict what comes next.

```
Input:  "def calculate_mean(numbers):"
          |
          v
   +-------------+
   |    LLM      |  (predicts next tokens based on patterns
   |             |   learned from billions of code examples)
   +-------------+
          |
          v
Output: "    return sum(numbers) / len(numbers)"
```

*ASCII created with Claude. It should be replaced with an image*

When applied to code, LLMs benefit from the fact that programming languages are highly structured and follow consistent patterns. The model doesn't "understand" code in the way humans do, but it has learned enough patterns to generate syntactically correct and often semantically meaningful code.

**Key insight**

LLMs are sophisticated pattern-matching systems, not reasoning engines. They excel at common patterns but can confidently produce incorrect code for novel or complex problems. **Always verify AI-generated code**.

**Practitioner's perspective: Simon Willison**

*"My current favorite mental model is to think of them as an over-confident pair programming assistant who's lightning fast at looking things up, can churn out relevant examples at a moment's notice and can execute on tedious tasks without complaint.*

*Over-confident is important. They'll absolutely make mistakes—sometimes subtle, sometimes huge. These mistakes can be deeply inhuman—if a human collaborator hallucinated a non-existent library or method you would instantly lose trust in them.*

*Don't fall into the trap of anthropomorphizing LLMs and assuming that failures which would discredit a human should discredit the machine in the same way."*

— Simon Willison, "How I use LLMs to help me write code"

## How are code LLMs trained?

Training a code LLM involves two main phases:

### 1. Pre-training

The model is exposed to massive amounts of code (and often natural language) to learn general programming patterns:

| Model | Training Data Size | Languages |
|---|---|---|
| StarCoder | 1 trillion tokens | 80+ languages |
| StarCoder2 | 3.3-4.3 trillion tokens | 600+ languages |
| CodeLlama | Llama 2 base + code | Multiple |
| GPT-4 / Claude | Undisclosed | Multiple |

During pre-training, the model learns:

- Syntax rules for various programming languages
- Common coding patterns and idioms
- Relationships between code and comments/documentation
- How different parts of a codebase relate to each other

### 2. Fine-tuning and instruction tuning

After pre-training, models are often further refined:

- **Fine-tuning**: Training on specific domains (e.g., scientific Python)
- **Instruction tuning**: Teaching the model to follow human instructions
- **RLHF** (Reinforcement Learning from Human Feedback): Aligning outputs with human preferences

## Training cut-off dates matter

A crucial characteristic of any model is its **training cut-off date**—the date at which training data collection stopped. For coding, this has direct practical implications:

| Impact | Example |
|---|---|
| Unknown libraries | A library released after the cut-off won't be suggested |
| Breaking changes | Major API changes since cut-off produce outdated suggestions |
| Deprecated patterns | Old syntax or methods may still be recommended |
| Security updates | Known vulnerabilities patched after cut-off won't be reflected |

**Key insight**

The training cut-off date is sometimes not known. With chatbots, you can try asking the chatbot. What makes it even more challenging is that some AI systems also have "tools" attached to them, so they can fetch **some** up-to-date content, while mixing with what they have seen during their training. This can be a total success, or total disaster. Pick a stable library, even if it is a little bit older. TODO: add link to https://boringtechnology.club/

**Practical implications:**

- Check model documentation for training cut-off dates
- Be skeptical of suggestions for rapidly-evolving libraries
- Provide recent documentation or examples in prompts when using newer tools
- Consider library stability as a factor in technology choices

## Open datasets for training code LLMs

Understanding what data models are trained on helps us understand their capabilities and limitations.

**Warning**

The summaries below were genrated with Claude. They need to be manually verified.

### The Stack (BigCode Project)

The Stack is a major open dataset for training code models:

| Version | Size | Languages | Key Features |
|---|---|---|---|
| v1 | 6.4 TB | 358 | Permissive licenses only |
| v2 | 67.5 TB | 600+ | Includes PRs, notebooks, docs |

**Important characteristics:**

- Only includes permissively-licensed code (MIT, Apache 2.0, etc.)
- Copyleft licenses (GPL) are excluded
- Provides an opt-out mechanism for developers who don't want their code included
- Built on the Software Heritage archive

**Discussion**

**Why does training data matter?**

Consider these implications:

- Models may reproduce patterns from their training data, including bugs
- Code style and conventions reflect the average of the training set
- Rare languages or niche libraries may have poor coverage
- The model cannot know about code written after its training cutoff

## The landscape of AI coding tools

AI coding assistants come in many forms. Here's an overview of the major categories and tools:

### Chatbots (Scenario I in this course)

General-purpose AI assistants accessed via web interface:

| Tool | Provider | Key Features |
|---|---|---|
| Duck.ai | DuckDuckGo | Privacy-focused, no account needed, free |
| ChatGPT | OpenAI | GPT-4, web browsing, code interpreter |
| Claude | Anthropic | Large context window, artifacts |
| Gemini | Google | Multimodal, Google integration |

**Recommended for exercises: Duck.ai**

For the exercises in this course, we recommend Duck.ai by DuckDuckGo. DuckDuckGo is known for its privacy-first philosophy—their search engine doesn't track users, and this extends to their AI chat service.

**Why Duck.ai for learning:**

- **No account required**: Start immediately without signup
- **Privacy by design**: Your IP address is hidden from AI providers, chats are not used for training, and conversations are stored locally on your device (not on remote servers)
- **Free access**: Includes Claude, GPT-4o mini, Llama, and Mistral models
- **Anonymous**: DuckDuckGo proxies your requests so AI providers never see your identity

This makes it ideal for experimenting with AI coding assistance without creating accounts or worrying about data retention policies. You can also access it via duckduckgo.com/chat or by typing `!ai` in DuckDuckGo search.

**How coding with a chatbot works:**

```
+--------+    copy/paste     +-----------+
|  You   | ---------------> | Chatbot   |
|        | <--------------- | (Web UI)  |
+--------+    copy/paste     +-----------+
     |
     v
+-----------+
| Your IDE  |
| (VS Code, |
|  Jupyter) |
+-----------+
```

**ASCII generated by Claude, it needs replacing with a block diagram**

## IDE-integrated assistants (Scenario II)

**TODO: This section needs expanding with more tools**

Tools that integrate directly into your development environment:

| Tool | Pricing | Key Features |
|---|---|---|
| GitHub Copilot | €XX/mo (free tier available) | Most mature, broad language support |
| Amazon Q Developer | Free for individuals | AWS expertise, security scanning |
| Codeium | Free core features | 70+ languages, Windsurf IDE |
| Tabnine | Free basic tier | Privacy-focused, local models available |

**How IDE-integrated AI assistant works:**

```
+------------------+          +------------------+
|    Your IDE      |  <---->  | Remote AI Server |
| (with extension) |          | (processes your  |
|                  |          |  code context)   |
+------------------+          +------------------+
```

## Agentic coding tools (Scenario III)

**TODO: this recently changed in early february so it needs to be updated**

Autonomous agents that can write, test, and modify code:

| Tool | Provider | Key Features |
|---|---|---|
| Claude Code | Anthropic | CLI-based, git integration, sandboxing |
| OpenAI Codex | OpenAI | CLI-based, git integration, sandboxing |
| GitHub Copilot Workspace | GitHub | PR-based workflow |
| Cursor | Cursor Inc. | AI-native IDE with agent mode |
| Aider | Open source | Terminal-based, multiple model support |

**How coding with agentic tools work:**

```
+--------+   high-level task   +------------+
|  You   | ------------------> |   Agent    |
|        |                     |            |
+--------+                     +-----+------+
                                     |
         +---------------------------+---------------------------+
         |                           |                           |
         v                           v                           v
  +-----------+               +-----------+               +-----------+
  | Read/Edit |               |   Run     |               |   Git     |
  |   Files   |               | Commands  |               | Operations|
  +-----------+               +-----------+               +-----------+
```

**The full spectrum is actually much broader**

These three scenarios are a teaching simplification. In reality, AI coding tools exist on a spectrum with many hybrids: PR-native agents, review bots, orchestration tools, local-first options, and more. See appendix-spectrum for an extended taxonomy.

## Current adoption and trends

TODOO: ADD HERE STATS ON ADOPTION AND TRENDS

## Limitations to keep in mind

Before diving into specific scenarios, remember these fundamental limitations:

### What LLMs cannot do

A typical generative AI system based on LLMs, without `toolcall` capabilities, cannot:

1. **Verify their own output**: They cannot run code or check if it works.
2. **Access real-time information**: Knowledge is frozen at training cutoff
3. **Understand your specific context**: They don't know your data, infrastructure, or requirements unless you tell them
4. **Guarantee correctness**: They optimize for "plausible" not "correct"

### Common failure modes

- **Hallucinated packages**: Suggesting libraries that don't exist
- **Outdated APIs**: Using deprecated functions or old syntax
- **Subtle bugs**: Code that looks right but has edge-case failures
- **Security vulnerabilities**: Not considering injection, authentication, etc.

**✍️ Exercise: Explore an AI chatbot**

Go to duck.ai (no account needed) and try the following:

1. Ask: "Write a Python function to calculate the standard deviation of a list"
2. Look at the response. Does it:
   - Use a built-in library or implement from scratch?
   - Handle edge cases (empty list, single element)?
   - Include documentation?
3. Now ask: "What assumptions does this code make? What could go wrong?"

**✔ Solution**

The exercise demonstrates several things:

- AI can generate working code quickly
- The code may not match your specific needs (maybe you wanted NumPy, or maybe you wanted the formula from scratch)
- Follow-up questions help identify limitations
- You should always review and test AI-generated code

## Summary

In this section, we covered the foundations of AI-assisted coding:

- LLMs generate code by predicting tokens based on patterns learned from training data
- Open datasets like The Stack provide transparent training data for code models
- Tools range from simple chatbots to fully autonomous agents
- Understanding limitations is crucial for responsible use

## See also

**TODO: THIS NEEDS TO BE REVISED**

- StarCoder: A State-of-the-Art LLM for Code - BigCode project blog
- The Stack v2 Paper - Technical details on training data
- BigCode Project - Open scientific collaboration on code LLMs
- Awesome-Code-LLM - Curated list of code LLM resources
- Simon Willison: How I use LLMs to help me write code - Practical insights from an experienced practitioner

**❗ Keypoints**

- LLMs are pattern-matching systems trained on billions of lines of code
- Training data (like The Stack) influences what models know and how they behave
- Tools range from chatbots (high control) to agents (low control)
- Always verify AI-generated code, as models cannot guarantee correctness

# Scenario I: Full Control (Chat-Based Coding)

**❓ Questions**

- How can I use generative AI chatbots effectively for coding tasks?
- What are the risks and benefits of manual copy/paste workflows?
- What prompting strategies work best for research code?

**❗ Objectives**

- Learn effective prompting techniques for code generation

- Understand what information is shared when using chatbots
- Develop a modular approach to building research pipelines with AI assistance
- Practice critical evaluation of AI-generated code

## The full control scenario

In this scenario, you interact with an AI assistant through a web interface (like ChatGPT, Claude, or Gemini) and manually copy code between the chatbot and your development environment.

```
+------------------+                    +------------------+
|                  |   Your prompts     |                  |
|    AI Chatbot    | <----------------- |       You        |
|    (Web UI)      |                    |                  |
|                  | -----------------> |                  |
+------------------+   Generated code   +--------+---------+
                                                 |
                                        manual   |
                                       copy/paste|
                                                 v
                                        +--------+---------+
                                        |    Your IDE      |
                                        | (VS Code/Jupyter)|
                                        |                  |
                                        | - Review code    |
                                        | - Run manually   |
                                        | - Debug yourself |
                                        +------------------+
```

**ASCII chart generated with Claude, it should be replaced with a block diagram**

### Why this is the lowest-risk approach

1. **You see everything**: Every piece of code goes through your eyes and clipboard
2. **Nothing runs automatically**: You decide when and how to execute code
3. **Clear boundaries**: The AI cannot access your files, run commands, or modify anything
4. **Explicit data sharing**: You control exactly what context the AI receives

## What information leaves your machine?

When you use a chat interface, the following is typically sent to the provider's servers:

| What you send | Considerations |
|---|---|
| Your prompts | May contain sensitive project details |
| Code you paste | Could include proprietary algorithms |
| Error messages | May reveal system configuration, usernames, local paths, filenames |

| What you send | Considerations |
|---|---|
| Data samples | Never paste real research data! |

**Data handling policies vary**

Different providers have different policies:

- Some use your conversations to train future models
- Some offer enterprise tiers with data isolation (e.g. Microsoft Azure)
- Read the terms of service for your chosen tool
- When in doubt, assume your input may be retained
- Most providers still retain all your activities for 30 days, even if you opted out.

### What NOT to share

- Real research data (use synthetic examples instead)
- API keys, passwords, or credentials
- Proprietary algorithms or research/trade/IPR secrets
- Patient/participant identifiable information
- Unpublished results or embargoed data

## Three modes of AI-assisted coding

Effective AI-assisted coding involves switching between three distinct modes depending on where you are in the development process.

### Exploration mode: Ask for options

When starting a project or evaluating approaches, use the AI for research:

```
"What are options for parsing XML in Python? Include pros/cons of each."

"What approaches could I use for parallelizing this computation?"

"What are some useful drag-and-drop libraries in JavaScript?
Build me an example demonstrating each one."
```

This helps you understand the landscape before committing to an approach. The AI's training cut-off means newer libraries won't be suggested, but for established options this is often fine—you want stability anyway.

### Planning mode: Design the workflow

Once you have explored your options, good planning makes sure you have the workflow sketched out, dependencies are considered, and your system has the right setup for starting the actual work (folder strcuture, version control, environment and other dependencies).

If the first stage was about brainstorming, this is about project management. A good well written plan makes it easier later for adding or removing components in your workflow. This is the equivalent of writing down all the comments, before writing your code.

Example of useful prompts:

TODO ENRICO TO FINISH THIS

### Production mode: Tell them exactly what to do

Once you've decided on an approach, switch to authoritarian prompting. Provide precise specifications:

```
Write a Python function with this signature:

def validate_participant_id(pid: str) -> bool:
    """
    Validate a participant ID matches our format: P followed by 3 digits.

    Parameters
    ----------
    pid : str
        The participant ID to validate

    Returns
    -------
    bool
        True if valid, False otherwise

    Raises
    ------
    TypeError
        If pid is not a string
    """
```

You act as the function designer; the AI implements to your specification.

**Practitioner's perspective: Function signatures as prompts**

*"I find LLMs respond extremely well to function signatures. I get to act as the function designer, the LLM does the work of building the body to my specification.*

*If your reaction to this is 'surely typing out the code is faster than typing out an English instruction of it', all I can tell you is that it really isn't for me any more. Code needs to be correct. English has enormous room for shortcuts, and vagaries, and typos, and saying things like 'use that popular HTTP library' if you can't remember the name off the top of your head.*

*The good coding LLMs are excellent at filling in the gaps. They're also much less lazy than me—they'll remember to catch likely exceptions, add accurate docstrings, and annotate code with the relevant types."*

— Simon Willison

## Managing context effectively

The key skill in AI-assisted coding is managing **context**—the accumulated text in your conversation that influences the AI's responses.

### What goes into context

| Element | Impact |
|---|---|
| Your prompts | Directly shape the response |
| AI's previous responses | Become part of the "memory" |
| Code you paste | Provides examples and patterns |
| Error messages shared | Help with debugging |

**TODO insert a box about context windows in popular AI systems, also talk about "memory" features which can influence the context**

### Practical context management

**Tips for context management:**

1. **Start fresh when stuck**: If a conversation isn't productive, begin a new one
2. **Build iteratively**: Get simple versions working, then add complexity
3. **Seed with examples**: Paste working code as context for new but similar tasks
4. **Provide multiple examples**: "Here are three similar functions I've written. Use them as inspiration for this new one."

## Effective prompting strategies

The quality of AI-generated code depends heavily on how you ask for it.

**TODO: this is a collection of promopts in no particilar order from an initial list expanded by claude. It needs refining, citations, sorting, what works well…**

### Strategy 1: Start with architecture, not implementation

Instead of asking for 500 lines of code at once, begin with structure:

**Poor approach:**

> "Write a complete Python script to analyze my fMRI data, including preprocessing, statistical analysis, and visualization."

**Better approach:**

> "I need to build a pipeline for fMRI analysis. What would be a good modular structure? I'm thinking separate functions for:
>
> 1. Loading data
> 2. Preprocessing
> 3. Statistical analysis
> 4. Visualization
>
> Can you outline what each module should do and what the interfaces between them should look like?"

### Strategy 2: Provide context about your environment

Help the AI understand your constraints:

```
I'm working on:
- Python 3.11
- Ubuntu 22.04
- Using NumPy, SciPy, and Matplotlib (prefer not to add new dependencies)
- Data is in HDF5 format

I need a function to...
```

### Strategy 3: Ask for explanations, not just code

```
Write a function to perform bootstrap resampling. Please:
1. Explain the statistical principle first
2. Show the implementation
3. Include comments explaining each step
4. Note any assumptions the code makes
```

### Strategy 4: Iterate incrementally

```
Session flow:
1. "Write a function to load CSV data with error handling"
2. [Test it, find an issue]
3. "This fails when the file has a BOM. Can you fix that?"
4. [Works now]
5. "Now let's add support for gzipped files..."
```

💬 Discussion

**Why modular development matters even more with AI**

When you generate code in small, testable pieces:

- You can verify each piece works before moving on
- Errors are easier to localize and fix
- You maintain understanding of what each part does
- The AI has clearer context for each request

### Strategy 5: Remember it's a conversation

The first response is rarely the final answer. Use follow-up prompts freely:

- "Break that repetitive code out into a function"
- "Use string manipulation methods rather than a regular expression"
- "Write that better!"
- "Now write tests for this using pytest"
- "Can you explain what line 15 does?"

## Building a research pipeline: Worked example

Let's walk through how to build a simple analysis pipeline using the chat-based approach.

### Step 1: Define the task

> "I have a CSV file with columns: participant_id, condition, reaction_time, accuracy. I need to:
>
> 1. Load and validate the data
> 2. Calculate summary statistics per condition
> 3. Run a statistical test comparing conditions
> 4. Create a visualization
>
> Let's start with step 1. Write a Python function to load the CSV and validate that all expected columns are present."

### Step 2: Review and test

The AI might respond with:

```python
import pandas as pd

def load_and_validate_data(filepath):
    """
    Load experimental data and validate required columns.

    Parameters
    ----------
    filepath : str
        Path to the CSV file

    Returns
    -------
    pd.DataFrame
        Validated dataframe

    Raises
    ------
    ValueError
        If required columns are missing
    """
    required_columns = {'participant_id', 'condition', 'reaction_time', 'accuracy'}

    df = pd.read_csv(filepath)

    missing = required_columns - set(df.columns)
    if missing:
        raise ValueError(f"Missing required columns: {missing}")

    return df
```

**Your review checklist:**

- Does it handle the file not existing?
- What about encoding issues?
- Is the error message helpful?

### Step 3: Iterate based on testing

> "This works, but I'd like to add:
>
> 1. Check for missing values in critical columns
> 2. Validate that reaction_time is positive
> 3. Log warnings instead of failing for recoverable issues"

### Step 4: Continue to next module

Once satisfied with data loading, move to summary statistics:

> "Great, that function is working. Now write a function that takes the validated dataframe and returns summary statistics (mean, std, n) for reaction_time and accuracy, grouped by condition."

## Common pitfalls and how to avoid them

### Pitfall 1: Accepting code without understanding it

**Problem**: You paste code and it works, but you don't understand why.

**Solution**: Always ask "Can you explain how this works?" If you can't explain the code to a colleague, you shouldn't use it.

### Pitfall 2: Not testing edge cases

**Problem**: Code works for your test case but fails on real data.

**Solution**: Explicitly ask about edge cases:

> "What edge cases should I test for this function? What inputs might break it?"

### Pitfall 3: Accumulating technical debt

**Problem**: Quick solutions that are hard to maintain long-term.

**Solution**: Ask for production-quality code:

> "Write this function following best practices: include docstrings, type hints, and error handling."

### Pitfall 4: Hallucinated packages or functions

**Problem**: AI suggests importing a package that doesn't exist.

**Solution**: Before running `pip install`, verify the package exists:

- Check on PyPI
- Search GitHub for the repository
- Be especially suspicious of packages you've never heard of

## When to relax the rules: Learning and exploration

The careful, verify-everything approach described above is essential for production code. But for **learning and exploration**, a looser approach can accelerate your understanding of what AI can do.

**Practitioner's perspective: Vibe-coding for learning**

Andrej Karpathy coined the term "vibe-coding" for a more relaxed approach:

*"There's a new kind of coding I call 'vibe coding', where you fully give in to the vibes, embrace exponentials, and forget that the code even exists. […] I ask for the dumbest things like 'decrease the padding on the sidebar by half' because I'm too lazy to find it. I 'Accept All' always, I don't read the diffs anymore."*

Simon Willison's take:

*"Andrej suggests this is 'not too bad for throwaway weekend projects'. It's also a fantastic way to explore the capabilities of these models—and really fun. The best way to learn LLMs is to play with them. Throwing absurd ideas at them and vibe-coding until they almost sort-of work is a genuinely useful way to accelerate the rate at which you build intuition for what works and what doesn't."*

**When vibe-coding is appropriate:**

- Learning a new library or framework
- Prototyping ideas quickly
- Building throwaway demonstrations
- Exploring what's possible

**When to be rigorous:**

- Research code that will be published
- Code that processes real data
- Anything that will be shared or maintained
- Security-sensitive applications

**Warning**

The distinction matters: vibe-coding is for **learning**, not for **production**. Don't let the ease of accepting suggestions lead you to deploy code you don't understand.

## Using AI to understand existing code

One of the lowest-risk, highest-value use cases for AI: asking it to explain code you didn't write.

### Why this is valuable for researchers

- Onboarding to inherited codebases
- Understanding library internals
- Code review preparation
- Learning from others' implementations

### Workflow for code understanding

1. **Paste the code** (or relevant portions) into a chat
2. **Ask for an overview**: "Give me an architectural overview of this code"
3. **Drill down**: "Explain what this specific function does"
4. **Identify issues**: "What could go wrong with this implementation?"

**Practitioner's perspective: Questions are low-stakes**

*"If the idea of using LLMs to write code for you still feels deeply unappealing, there's another use-case for them which you may find more compelling. Good LLMs are great at answering questions about code.*

*This is also very low stakes: the worst that can happen is they might get something wrong, which may take you a tiny bit longer to figure out. It's still likely to save you time compared to digging through thousands of lines of code entirely by yourself.*

*The trick here is to dump the code into a long context model and start asking questions."*

— Simon Willison

### Example prompts for code understanding

```
"What is the main purpose of this module?"

"Trace the data flow from input to output."

"What external dependencies does this code have?"

"Where might this code fail with unexpected input?"

"Explain the algorithm in step 3 of this function."
```

This is a great starting point for researchers who are hesitant about AI-generated code—you remain in full control, and you're using the AI to augment your understanding rather than replace your work.

## Exercises

**✍ Exercise Chat-1: Build a data loader**

Using duck.ai (or your preferred AI chatbot):

1. Ask it to write a function that loads JSON files containing experimental results with this structure:

   ```
   {"participant": "P01", "scores": [85, 90, 78], "completed": true}
   ```

2. Ask it to add validation for:

    - Required fields
    - Score values between 0-100
    - Proper data types
3. Test the resulting function with valid and invalid inputs
4. Ask the AI: “What could still go wrong with this function?”

**✓ Solution**

The key learning points:

- Starting with structure before asking for implementation
- Iterating to add validation
- Explicitly asking about failure modes

Your final function should handle:

- Missing files
- Invalid JSON syntax
- Missing required fields
- Out-of-range values
- Wrong data types

**✍ Exercise Chat-2: Debug with AI assistance**

Take this buggy code and use duck.ai to help fix it:

```python
def calculate_statistics(data):
    mean = sum(data) / len(data)
    variance = sum((x - mean) ** 2 for x in data) / len(data)
    std = variance ** 0.5
    return {"mean": mean, "std": std, "variance": variance}

# This crashes:
result = calculate_statistics([])
```

Practice:

1. Describe the error to the chatbot
2. Ask it to fix the bug
3. Ask what other edge cases might cause problems

**✓ Solution**

The function fails on empty lists (division by zero). A proper fix handles:

- Empty lists
- Single-element lists (variance undefined or requires Bessel's correction)
- The difference between population and sample variance

Good prompting would reveal these statistical subtleties.

## Best practices checklist

When using chat-based AI coding:

- [ ] **Never paste sensitive data** - use synthetic examples
- [ ] **Start with architecture** - outline before implementation
- [ ] **Work modularly** - small functions you can test
- [ ] **Verify packages exist** - check PyPI before installing
- [ ] **Test thoroughly** - include edge cases
- [ ] **Understand the code** - if you can't explain it, don't use it
- [ ] **Document your process** - note which parts were AI-generated

## The real benefit: Enabling more ambitious projects

It's tempting to think of AI coding assistants purely in terms of speed. But the deeper benefit is different.

**Practitioner's perspective: Speed enables ambition**

*"This is why I care so much about the productivity boost I get from LLMs: it's not about getting work done faster, it's about being able to ship projects that I wouldn't have been able to justify spending time on at all.*

*The fact that LLMs let me execute my ideas faster means I can implement more of them, which means I can learn even more."*

— Simon Willison

For researchers, this might mean:

- Building a quick visualization tool you wouldn't have time to code manually
- Prototyping analysis approaches before committing to one
- Creating small utilities that improve your workflow
- Exploring "what if" questions through rapid implementation

**Warning**

**Managing expectations**: LLMs amplify existing expertise. An experienced developer will get better results than a beginner because they:

- Know what to ask for
- Can evaluate whether the output is correct
- Understand which follow-up questions to ask
- Recognize when the AI is wrong

This doesn't mean beginners can't benefit—they can. But don't expect AI to instantly give you expert-level results if you're still learning the fundamentals.

## Summary

The chat-based approach offers maximum control over AI-assisted coding:

- You explicitly choose what information to share
- You review all code before it runs
- You maintain full understanding of your codebase

The trade-off is speed: manually copying code and context takes time. For many research applications, this trade-off is worthwhile, as the control and transparency support reproducibility and scientific integrity.

## See also

- Prompting Guide - General prompting techniques
- Google: Best Practices for AI Coding Assistants
- Simon Willison: How I use LLMs to help me write code - Detailed practical workflow
- Zero To Mastery: How to Use ChatGPT to 10x Your Coding - Prompt engineering techniques

**Keypoints**

- Chat-based AI coding gives you maximum control over data and execution
- Use **exploration mode** (ask for options) when researching, **production mode** (precise specs) when implementing
- Context management is the key skill: start fresh when stuck, build iteratively, provide examples
- Work in small, testable modules rather than generating large code blocks
- Iterate freely: the first response is a starting point, not the final answer
- AI is also valuable for understanding existing code, not just generating new code
- The real benefit is enabling projects you wouldn't otherwise have time for
- Never share sensitive research data with AI chatbots

## Scenario II: IDE Integration (Tab Completion and Inline Suggestions)

**❓ Questions**

- How do IDE-integrated AI assistants like GitHub Copilot work?
- What information is sent to remote servers when I use these tools?
- How can I configure these tools to balance productivity and privacy?

**❗ Objectives**

- Understand how IDE-integrated AI assistants process your code
- Learn to configure GitHub Copilot and similar tools
- Recognize the reduced transparency compared to chat-based approaches
- Develop habits for reviewing inline suggestions critically

### The IDE integration scenario

In this scenario, AI assistance is integrated directly into your development environment. As you type, the AI suggests completions, entire functions, or even multi-line code blocks.

```
+-----------------------------------------------------+
|  Your IDE (VS Code, JetBrains, etc.)                |
|                                                     |
|  +-----------------------------------------------+  |
|  | def calculate_mean(numbers):                  |  |
|  |     """Calculate the arithmetic mean."""      |  |
|  |     return sum(numbers) / len(numbers)        |  | <-- AI suggests this
|  |            ~~~~~~~~~~~~~~~~~~~~~~~~~~~~       |  |     as you type
|  |            [Tab to accept] [Esc to dismiss]   |  |
|  +-----------------------------------------------+  |
|                                                     |
+-------------------------+---------------------------+
                          |
                          | Code context sent
                          | automatically
                          v
              +----------+----------+
              |   Remote AI Server  |
              |                     |
              |  - Processes your   |
              |    file contents    |
              |  - Returns          |
              |    suggestions      |
              +---------------------+
```

**ASCII diagram generated with claude, we should use screenshots here**

### How this differs from chat-based coding

| Aspect | Chat-based (Scenario I) | IDE-integrated (Scenario II) |
|---|---|---|
| **Data flow** | You explicitly paste code | Context sent automatically |
| **Visibility** | You see exactly what you share | Less clear what's transmitted |
| **Execution** | Manual copy/paste | Still manual, but faster |
| **Context** | Limited to what you provide | May include entire project |

## What information leaves your machine?

When you use IDE-integrated tools, data transmission happens automatically and continuously. Understanding what's sent is crucial.

### Typically transmitted data

| Data Type | Purpose | Privacy Concern |
|---|---|---|
| Current file content | Generate relevant suggestions | May contain sensitive code |
| Open file tabs | Broader context | More exposure |
| File paths | Understand project structure | Reveals project organization |
| Cursor position | Know what to complete | Minimal concern |
| Recent edits | Understand your intent | Sent frequently |

### What may be transmitted (tool-dependent)

- Other files in your workspace
- Git history
- Comments and docstrings
- Import statements (revealing your dependencies)

**Read your tool's documentation**

Each tool has different data collection policies. For example:

- GitHub Copilot transmits code context to GitHub/Microsoft servers
- Some tools offer "local only" models (e.g., Tabnine)
- Enterprise tiers may offer data isolation guarantees

## Setting up GitHub Copilot

**THIS MIGHT HAVE CHANGED IN EARLY FEBRUARY, IT NEEDS VERIFICATION PLUS LINKS**

GitHub Copilot is currently the most widely-used IDE-integrated assistant. Here's how to set it up with privacy and control in mind.

### Installation in VS Code

1. Install the "GitHub Copilot" extension from the marketplace
2. Sign in with your GitHub account
3. If you have an educational (.edu) email, you may qualify for free access

### Key configuration settings

Open VS Code settings (Ctrl+, or Cmd+,) and search for "Copilot":

```
{
    // Disable automatic suggestions (require manual trigger)
    "github.copilot.enable": {
        "*": true,
        "markdown": false,  // Disable for documentation
        "plaintext": false  // Disable for plain text files
    },

    // Require explicit trigger instead of automatic suggestions
    "editor.inlineSuggest.enabled": true,

    // Control which files Copilot can access
    "github.copilot.advanced": {
        "indentationMode": {
            "python": true,
            "javascript": true
        }
    }
}
```

### Disabling for sensitive files

You can create a `.copilotignore` file in your project root:

```
# Don't process these files
.env
secrets.py
config/credentials.yaml
data/*  # Ignore data directory
```

💬 Discussion

**When to disable AI suggestions**

Consider disabling inline AI for:

- Files containing credentials or API keys
- Proprietary algorithms
- Files with sensitive personal data
- Configuration files with internal infrastructure details

## The hidden context problem

A key difference from chat-based AI is that you don't explicitly control what context the model receives. This has implications:

### Example: Unintended context sharing

```
+-------------------+     +-------------------+     +-------------------+
|   main.py         |     |   config.py       |     |   secrets.env     |
|   (you're here)   |     |   (open tab)      |     |   (open tab!)     |
+-------------------+     +-------------------+     +-------------------+
         |                        |                        |
         +------------------------+------------------------+
                                  |
                                  v
                      All may be sent as context
```

**The issue**: You might have a secrets file open in another tab, and parts of it could be included in the context sent to the AI server.

### Mitigation strategies

1. **Close sensitive tabs** when using AI completion
2. **Use** `.copilotignore` to exclude sensitive files
3. **Consider workspace separation** - keep sensitive projects in different VS Code windows
4. **Review your open tabs** before intensive AI-assisted sessions

## Features beyond simple completion

Modern IDE integrations offer more than tab completion:

### Inline chat

Many tools now allow you to chat with the AI directly in your editor:

- Select code and ask "explain this"
- Ask to refactor or improve selected code
- Generate tests for highlighted functions

### Automated multi-file edits

Some tools (like Cursor, Copilot Workspace) can:

- Edit multiple files simultaneously
- Apply changes across your codebase
- Suggest refactoring patterns

**Increased risk with automated edits**

When tools can modify multiple files:

- Changes may have unintended consequences
- Harder to review all modifications
- Version control becomes essential

**Always commit before automated multi-file operations.**

## Developing critical review habits

With suggestions appearing automatically, it's easy to accept them without sufficient review. Build these habits:

### The 3-second rule

**TODO ENrico has a reference for this**

Before pressing Tab to accept:

1. **Read** the entire suggestion
2. **Ask** yourself: "Does this match my intent?"
3. **Check** for obvious issues (wrong variable names, edge cases)

### Watch for common issues

| Issue | Example | How to Catch |
|---|---|---|
| Wrong variable | `return total / len(items)` when you named it `data` | Check names match your code |
| Missing edge case | No check for empty list | Ask yourself about inputs |
| Outdated API | Using deprecated function | Check documentation |
| Wrong assumption | Assuming sorted input | Read comments critically |

### Accept-then-review workflow

Some developers find it faster to:

1. Accept the suggestion
2. Immediately review line-by-line
3. Delete and redo if wrong

This works but requires discipline. Don't skip step 2.

## Alternative tools: Codeium and Tabnine

GitHub Copilot isn't your only option. Here's how alternatives compare:

### Codeium

- **Pricing**: Free core features
- **Privacy**: Claims no training on your code
- **Features**: 70+ languages, chat interface, Windsurf IDE

Setup in VS Code:

1. Install "Codeium" extension
2. Create account at codeium.com
3. Authenticate in VS Code

### Tabnine

- **Pricing**: Free basic tier, paid for advanced
- **Privacy**: Offers local-only models (no cloud transmission)
- **Features**: Learns from your codebase, SOC 2 compliant

Setup in VS Code:

1. Install "Tabnine" extension
2. Create account
3. Choose cloud or local mode

### Comparison for research settings

| Factor | Copilot | Codeium | Tabnine (Local) |
|---|---|---|---|
| Privacy | Cloud-based | Cloud-based | Can run locally |
| Quality | Excellent | Good | Good |
| Cost | $10-19/mo | Free | Free basic |
| Institutional acceptance | Varies | Varies | Often preferred |

## Exercises

### ✍️ Exercise IDE-1: Configure your tool

Set up GitHub Copilot (or Codeium/Tabnine) and:

1. Create a `.copilotignore` or equivalent to exclude:
    - Any `.env` files
    - A `secrets/` directory
    - Data files (`*.csv`, `*.json` in `data/`)
2. Find the setting to disable suggestions for Markdown files
3. Test that suggestions work in a Python file but not in the excluded files

### ✓ Solution

Example `.copilotignore`:

```
# Sensitive files
.env
.env.*
secrets/
credentials.*

# Data files
data/
*.csv
*.json
*.parquet

# Documentation (disable AI here)
*.md
docs/
```

In VS Code settings:

```
{
    "github.copilot.enable": {
        "*": true,
        "markdown": false
    }
}
```

### ✍️ Exercise IDE-2: Critical suggestion review

With AI suggestions enabled, type the following function signature and observe the suggestions:

```python
def validate_email(email_string):
    """Check if a string is a valid email address."""
```

1. Accept the suggestion
2. Review it critically:
   - Does it use regex or simple string checks?
   - Does it handle all valid email formats?
   - Is it too strict or too lenient?
3. Search online for email validation edge cases
4. Ask yourself: is the AI-generated solution appropriate for your use case?

### ✔ Solution

Common issues with AI-generated email validation:

- May reject valid emails (e.g., with + signs, subdomains)
- May accept invalid emails (missing TLD validation)
- Regex might be overly complex or overly simple

The key lesson: "valid email" is surprisingly complex, and AI suggestions reflect common patterns, not necessarily correct patterns.

For many applications, using a well-tested library (like Python's `email-validator`) is better than AI-generated regex.

### ✍ Exercise IDE-3: Context awareness experiment

Test what context your AI tool uses:

1. Open two Python files in VS Code:

   - `main.py`: Empty except for a function signature
   - `helpers.py`: Contains a utility function
2. In `helpers.py`, write:

   ```python
   def calculate_tax(amount, rate=0.21):
       """Calculate tax amount for the Netherlands."""
       return amount * rate
   ```

3. In `main.py`, start typing:

```python
def process_invoice(amount):
    tax = 
```

4. Does the AI suggest calling `calculate_tax`?
5. What does this tell you about cross-file context?

**✓ Solution**

If the AI suggests `calculate_tax`, it demonstrates that:

- Your open files are being used as context
- The AI understands relationships between files
- This is useful but also means sensitive code in open tabs could inform suggestions

This experiment shows why closing sensitive files matters when using IDE-integrated AI tools.

## When to use IDE integration vs. chat

| Use IDE integration when | Use chat-based when |
|---|---|
| Writing boilerplate code | Designing architecture |
| Completing obvious patterns | Debugging complex issues |
| Speed matters, risk is low | You need to control context carefully |
| Working on non-sensitive code | Learning a new concept |

## Summary

IDE-integrated AI assistants offer faster workflows than chat-based approaches, but with reduced visibility into what data is shared:

- Context is transmitted automatically
- You must proactively configure exclusions
- Critical review habits are essential
- Alternative tools offer different privacy trade-offs

The key question is: **Do you know what's being sent to the AI server?** If you're not sure, either find out or switch to a more transparent approach.

### See also

- GitHub Copilot Documentation
- Codeium Documentation
- Tabnine Documentation
- VS Code AI Extensions Comparison

**Keypoints**

- IDE-integrated AI sends code context automatically to remote servers
- Use configuration files (`.copilotignore`) to exclude sensitive files
- Always review suggestions before accepting, even if they look correct
- Consider local-model alternatives (Tabnine) for sensitive projects
- Close sensitive tabs when working with AI-assisted completion

# Scenario III: Full Agentic Code Development

**TODO this changed again in early february and needs updating…**

**❓ Questions**

- What are agentic coding tools and how do they differ from other approaches?
- What risks come with giving an AI agent access to your system?
- How can I use these tools while maintaining appropriate oversight?

**Objectives**

- Understand how agentic AI coding tools work
- Recognize the risks of autonomous code modification and command execution
- Learn about sandboxing and permission controls
- Develop strategies for supervising AI agents in coding tasks

## The agentic scenario

In this scenario, you give an AI agent a high-level task, and it autonomously writes code, executes commands, runs tests, and modifies files to accomplish the goal. You basically become a project manager rather than a coder; depending on the level of automation, you might not even see any of the code that is generated and run.

```
+--------+                              +---------------------+
|  You   |   "Add user authentication  |                     |
|        |    to this Flask app"       |    AI Agent         |
|        | -------------------------->  |                     |
+--------+                              |  Plans approach     |
    ^                                   |  Reads your files   |
    |                                   |  Writes new code    |
    |  "Done. I've added login,        |  Runs tests         |
    |   registration, and session      |  Installs packages  |
    |   management. Here's what        |  Makes git commits  |
    |   I changed..."                  |                     |
    +----------------------------------  +---------------------+
                                                 |
              +---------------------------------+---------------------------------+
              |                |                |                |                |
              v                v                v                v                v
        +---------+      +---------+      +---------+      +---------+      +---------+
        | Read    |      | Write   |      | Execute |      | Install |      | Git     |
        | Files   |      | Files   |      | Commands|      | Packages|      | Commit  |
        +---------+      +---------+      +---------+      +---------+      +---------+
```

**ASCII chart generated with Claude, let's make a nice block diagram**

### Why this is the highest-risk approach

1. **Autonomous execution**: Agent runs commands without per-command approval
2. **System access**: May have access to your shell, files, network
3. **Opaque decision-making**: Hard to predict what the agent will do
4. **Irreversible actions**: Deleted files, pushed commits, installed packages

## Landscape of agentic coding tools

### Claude Code (Anthropic)

**TODO: rewrite with the /plan command and mention skills and /learn commands**

A terminal-based agent that lives in your shell:

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

# Run
claude
```

**Capabilities:**

- Reads and writes files in your project
- Executes shell commands
- Understands git workflows
- Can work across multiple files

**Safety features:**

- Sandboxing via Bubblewrap (Linux) or Seatbelt (macOS)
- Asks for permission before potentially dangerous operations
- Shows proposed changes before applying

**Practitioner's perspective: A real Claude Code session**

Simon Willison documented a real project built with Claude Code—a colophon page showing the commit history of tools he'd built. The session took 17 minutes and cost $0.61 in API calls.

His prompting sequence:

1. "Build a Python script that checks the commit history for each HTML file and extracts any URLs from those commit messages…"
2. [Inspected output, found issues] "It looks like it just got the start of the URLs, it should be getting the whole URLs…"
3. "Update the script—I want to capture the full commit messages AND the URLs…"
4. "This is working great. Write me a new script called build_colophon.py…"
5. [Got wrong output] "It's not working right. ocr.html had a bunch of commits but there is only one link…"
6. "It's almost perfect, but each page should have the commits displayed in the opposite order—oldest first"
7. "One last change—the pages are currently listed alphabetically, let's instead list them with the most recently modified at the top"

Key observations:

- Started with vibe-coding (didn't look at the code it wrote)
- Iterated based on observable output
- Gave specific, concrete feedback when things were wrong
- Let the agent handle implementation details

*"I was expecting that 'Test' job, but why were there two separate deploys? […] It was time to ditch the LLMs and read some documentation!"*

Even experienced practitioners reach points where they need to take over.

### OpenAI Codex

**REDO** Erased old stuff, need to check the updates.

### Cursor (AI-native IDE)

**TODO: this below is AI generated, I don't have cursor to test this**

A VS Code-based editor with deep AI integration:

**Agent mode:**

- Plans multi-file changes
- Executes changes autonomously
- Runs up to 8 parallel agents
- Shows combined diff views

**Safety features:**

- Sandboxed terminal by default
- Shows changes before committing
- SOC 2 compliant privacy mode

### Other tools

| Tool | Type | Key Characteristic |
|---|---|---|
| Aider | Terminal | Open source, multiple model support |
| GitHub Copilot Workspace | Web | PR-centric workflow |
| Continue | IDE extension | Open source, customizable |

## What can go wrong?

**TODO: add links to reddit and other horror stories**

Agentic tools can cause serious problems if not properly supervised.

### File system risks

```
Potential agent actions:
- Delete important files (rm, unlink)
- Overwrite configurations
- Modify files outside project directory
- Fill disk with generated content
```

**Real example**: Agents have been observed attempting commands like `rm -rf /` when confused about cleanup tasks.

### Command execution risks

```
Dangerous commands an agent might run:
- curl ... | bash          (arbitrary code execution)
- pip install <package>    (supply chain risk)
- chmod 777 ...            (security weakening)
- ssh, scp                 (network access)
```

### Data exfiltration risks

```
Ways agents could leak data:
- Include in API requests to the AI service
- Execute curl/wget to external URLs
- Write to network-accessible locations
- Include in git commits pushed to public repos
```

### Package installation risks (Hallucination attacks)

AI models sometimes suggest packages that don't exist. Attackers can:

1. Monitor AI suggestions for hallucinated package names
2. Register those packages on PyPI/npm with malicious code
3. Wait for developers to install them

This is called "slopsquatting":

```
AI suggests:    pip install flask-security-utils
Reality:        This package doesn't exist... or does it now?
                An attacker may have registered it with malware
```

**Supply chain attacks are real**

The 2024 xz-utils backdoor showed how sophisticated supply chain attacks can be. Agentic tools that install packages without verification increase this attack surface significantly.

## Permission models and controls

Different tools offer different levels of control:

### Claude Code permission system

**TODO: this needs to be updated**

Claude Code asks for permission before certain operations:

```
Claude wants to run: pip install pandas
Allow? [y/n/always/never]
```

You can configure default permissions:

```
// .claude/settings.json
{
  "permissions": {
    "allow_file_read": true,
    "allow_file_write": "ask",
    "allow_shell_commands": "ask",
    "allow_network": false
  }
}
```

## Sandboxing levels

| Level | What it restricts | Tools |
|---|---|---|
| **None** | Full system access | Dangerous! |
| **Process** | Network, some syscalls | Bubblewrap, Seatbelt |
| **Container** | Filesystem, network | Docker |
| **VM** | Everything | Firecracker microVM |

## Practical sandboxing with Docker

Run your agent inside a container:

```
# Create a Dockerfile for your project
FROM python:3.11-slim

WORKDIR /project
COPY . .

# Install dependencies
RUN pip install -r requirements.txt

# Run agent inside container
```

```
# Run with limited access
docker run -it \
  --network none \          # No network access
  --read-only \             # Read-only root filesystem
  --tmpfs /tmp \            # Writable tmp only
  -v $(pwd):/project \      # Mount project directory
  my-agent-container
```

## Supervision strategies

If you choose to use agentic tools, implement these safeguards:

### 1. Start with read-only mode

Many tools have modes where they can only suggest, not execute:

```
# Claude Code in plan-only mode
claude --plan-only "Add error handling to main.py"
```

### 2. Review all proposed changes

```
Agent proposes changes to 3 files:
  [M] src/main.py      (+15, -3)
  [M] src/utils.py     (+8, -0)
  [A] tests/test_main.py (+45)

Review changes? [y/n]
```

**Never skip this step.** Read every change.

### 3. Use version control religiously

```
# Before any agentic session
git status                    # Check clean state
git stash                     # Stash any changes
git checkout -b ai-experiment # Work on a branch

# After agent finishes
git diff                      # Review all changes
git add -p                    # Stage selectively
git commit                    # Or git reset --hard to undo
```

### 4. Set up monitoring

Watch what the agent is doing:

```
# In another terminal, watch file changes
watch -n 1 'git status'

# Or use a file system watcher
fswatch -r . | while read f; do echo "Modified: $f"; done
```

### 5. Limit scope

Give agents minimal access:

```
Good: "Fix the bug in parse_data() function in src/parser.py"
Bad:  "Fix all the bugs in the project"

Good: "Add a unit test for the validate_email function"
Bad:  "Set up the entire test infrastructure"
```

## When agentic tools make sense (and when they don't)

### Appropriate use cases

- **Boilerplate generation**: "Create a Flask app skeleton"
- **Test generation**: "Write unit tests for this module"
- **Documentation**: "Generate docstrings for these functions"
- **Refactoring**: "Rename this variable across all files"
- **Exploration**: "Explain how this codebase handles authentication"

### Inappropriate use cases

- **Production deployments**: Too much can go wrong
- **Database migrations**: Need human review
- **Security-critical code**: Don't trust without verification
- **Code with sensitive data**: Risk of exposure
- **Unfamiliar codebases**: You can't verify what you don't understand

## Be ready to take over

Even with excellent tools, there comes a point where human expertise is needed.

**Practitioner's perspective: Know when to step in**

*"LLMs are no replacement for human intuition and experience. I've spent enough time with GitHub Actions that I know what kind of things to look for, and in this case it was faster for me to step in and finish the project rather than keep on trying to get there with prompts."*

— Simon Willison

His colophon project worked, but something wasn't right: there were two deploy jobs running. Rather than continue prompting, he read the documentation and found a settings switch that needed changing.

**The skill is knowing when:**

- More prompting won’t help
- You need to understand the underlying system
- Reading documentation is faster than iterating
- The AI is confidently going in the wrong direction

The most effective approach combines AI speed with human oversight—not blind trust in either direction.

## Exercises

### ✍️ Exercise Agent-1: Explore an agent (read-only)

If you have access to an agentic tool (Claude Code, Cursor, etc.):

1. Start it in read-only or plan-only mode
2. Give it a task: “Explain the structure of this project”
3. Observe:
    - What files does it try to read?
    - What is its reasoning process?
    - How does it present findings?
4. Do NOT allow it to make any changes yet

This helps you understand how the agent “thinks” before giving it write access.

### ✔ Solution

Key observations to make:

- The agent typically starts by reading project structure (ls, tree)
- It then reads key files (README, main entry points, config)
- It may make assumptions that are wrong about your project
- Its explanation reveals its understanding (which may have gaps)

This exploration builds trust incrementally before allowing modifications.

### ✍️ Exercise Agent-2: Sandboxed experiment

Set up a sandboxed environment to safely experiment with agentic tools:

1. Create a test project directory with some simple Python files
2. Initialize a git repository
3. If using Docker:

```
docker run -it --network none -v $(pwd):/project python:3.11 bash
```

4. If not using Docker, at minimum:
    - Work in a dedicated directory
    - Keep version control active
    - Don't give the agent access to sensitive directories
5. Try giving the agent a simple task and observe its behavior

**✓ Solution**

The exercise teaches:

- How to isolate experiments from important data
- The importance of version control as a safety net
- How agents behave when they have limited access
- The difficulty of truly sandboxing modern tools (they often need network access to function)

**✍️ Exercise Agent-3: Permission audit**

For an agentic tool you're considering:

1. Find its documentation on permissions and data handling
2. Answer these questions:
    - What data is sent to remote servers?
    - Can it be configured to work locally?
    - What system resources can it access?
    - How does it handle credentials in your project?
    - Is there an audit log of actions taken?
3. Based on your findings, what safeguards would you implement before using it?

**✓ Solution**

This exercise builds security awareness. Key findings typically include:

- Most agentic tools require network access to the AI provider
- Few offer fully local operation
- Audit logs are often incomplete
- Credential handling varies widely
- Documentation may not cover all data flows

Safeguards should include version control, network monitoring, and credential isolation.

## Comparison: When to use each scenario

| Factor | Scenario I (Chat) | Scenario II (IDE) | Scenario III (Agent) |
|---|---|---|---|
| **User control** | Full | Moderate | Limited |
| **Speed** | Slow | Fast | Fastest |
| **Visibility** | Complete | Partial | Low |
| **Risk level** | Low | Medium | High |
| **Best for** | Learning, design | Routine coding | Boilerplate, refactoring |
| **Avoid for** | Large volumes | Sensitive code | Production systems |

## Summary

Agentic coding tools offer the most automation but also the highest risk:

- They can read files, execute commands, and modify your codebase
- Risks include data exposure, malicious packages, and destructive actions
- Sandboxing and permission controls provide some protection
- Version control is your essential safety net
- Always review all changes before accepting them

The question isn't "Should I ever use agentic tools?" but rather "For which tasks, with what safeguards, and how much supervision?"

## See also

- Claude Code Documentation
- Cursor Documentation
- Docker Security Best Practices
- Bubblewrap (sandboxing tool)
- OpenSSF Guide for AI Code Assistants
- Simon Willison: Claude Code example with transcript - Real-world agentic session
- DataNorth: Claude Code Complete Guide - Comprehensive comparison with other tools

**Keypoints**

- Agentic tools can read, write, and execute without per-action approval
- Risks include file deletion, data exposure, and supply chain attacks
- Use sandboxing (Docker, Bubblewrap) to limit agent access
- Always use version control and review all changes
- Match tool autonomy to task risk, use agents for low-stakes tasks

# Quick Reference

A condensed reference for the lesson on responsible AI-assisted coding.

## The three scenarios

| Scenario | Control | Risk | Best for |
|---|---|---|---|
| I: Chat | Full | Low | Learning, design, sensitive work |
| II: IDE integration | Medium | Medium | Routine coding, boilerplate |
| III: Agentic | Limited | High | Refactoring, test generation |

## Security checklist

### Before any AI interaction

- [ ] Remove credentials from files
- [ ] Use synthetic data, not real data
- [ ] Close sensitive tabs (IDE integration)

### When AI suggests a package

```
# Verify it exists before installing
pip index versions <package-name>
# Or, even better, check https://pypi.org/project/<package-name>
```

### When AI generates code

- [ ] Read and understand all code
- [ ] Check for injection vulnerabilities (SQL, command, path)
- [ ] Test edge cases (e.g.: empty input, None, negative numbers)
- [ ] Run security linter like Bandit: `bandit -r your_code/`

## Effective prompting

### Two modes of prompting

| Mode | When | How |
|---|---|---|
| **Exploration** | Starting, researching | "What are options for doing X?" |
| **Planning** | Designing a stable workflow | "I want to do task X using Y and Z, please plan each script and function needed for designing the workflow." |

| Mode | When | How |
|---|---|---|
| Production | Implementing | Provide exact function signatures |

### Do: Be specific with function signatures

```
Write a Python function with this signature:

def validate_email(address: str) -> bool:
    """Validate email using a well-tested library.
    Return True if valid, False otherwise.
    Raise TypeError if address is not a string."""
```

### Don't: Be vague

```
"Write code to handle emails"
```

### Structure of prompts for coding

1. Context (programming language, libraries, environment)
2. Task (what you want)
3. Constraints (what to avoid, requirements)
4. Format (documentation, tests, explanations)

### Remember: It's a conversation

- First response is a starting point, not the answer
- Iterate: "Break that out into a function", "Split this long script into independent units"
- Ask follow-ups: "What could go wrong with this?"

## IDE integration settings

### VS Code settings.json

Example to make sure some configurations yaml files are not accessed by copilot (uncertain if copilot respects this).

```
{
        "github.copilot.enable": {
                "*": true,          // enable Copilot for all languages by default
                "yaml": false,      // disable for YAML files
                "plaintext": false  // disable for plain text files
        }
}
```

### .copilotignore

```
.env
.env.*
secrets/
credentials.*
*.pem
*.key
data/
```

## Sandboxing commands

### Docker (basic isolation)

TODO: Needs testing on multiple systems

Gets a basic python container and starts bash:

```
docker run -it \
    --network none \
    --read-only \
    --tmpfs /tmp \
    -v $(pwd):/project \
    python:3.11 bash
```

### Bubblewrap (Linux)

TODO: Needs testing and limitations on where it works

Limits the command “your_command” to certain places in your Linux computer.

```
bwrap \
    --ro-bind /usr /usr \
    --ro-bind /lib /lib \
    --bind $(pwd) $(pwd) \
    --unshare-net \
    your_command
```

## Common vulnerabilities in AI code

TODO: this list is not exhaistive. We should add more links / further readings.

### SQL Injection

```python
# Bad (AI often generates this)
cursor.execute(f"SELECT * FROM users WHERE id = {user_id}")

# Good
cursor.execute("SELECT * FROM users WHERE id = ?", (user_id,))
```

### Path Traversal

```python
# Bad
open(f"/data/{filename}")

# Good
from pathlib import Path
path = (Path("/data") / filename).resolve()
if not path.is_relative_to(Path("/data")):
    raise ValueError("Invalid path")
```

### Command Injection

```python
# Bad
os.system(f"ping {hostname}")

# Good
subprocess.run(["ping", "-c", "1", hostname], check=True)
```

## Package verification script

TODO: this was generated with Claude. To be tested.

```python
import requests

def verify_package(name):
    """Check if a package exists on PyPI."""
    r = requests.get(f"https://pypi.org/pypi/{name}/json")
    if r.status_code == 404:
        print(f"WARNING: {name} not found!")
        return False
    info = r.json()["info"]
    print(f"Found: {name}")
    print(f"  Author: {info.get('author')}")
    print(f"  Home: {info.get('home_page')}")
    return True
```

## Resources

- OWASP LLM Top 10
- OpenSSF AI Code Assistant Guide
- Bandit security linter

- PyPI package search
- Simon Willison: How I use LLMs for code
- Addy Osmani: My LLM coding workflow

# Instructor's guide

## Why we teach this lesson

AI coding assistants are already part of the daily workflow used by researchers and developers, but many users lack awareness of the security risks and best practices. This lesson fills a critical gap by:

- Providing a risk-aware framework rather than just tool tutorials
- Helping researchers make informed decisions about AI tool adoption
- Teaching security practices specific to AI-assisted coding
- Addressing the unique needs of researchers (data sensitivity, reproducibility, institutional policies)

## Intended learning outcomes

After this lesson, learners should be able to:

1. **Explain** how LLMs generate code and their fundamental limitations
2. **Compare** the three scenarios (chat, IDE, agentic) in terms of control and risk
3. **Configure** AI coding tools with appropriate privacy and security settings
4. **Verify** AI-suggested packages and review code for common vulnerabilities
5. **Implement** sandboxing strategies for agentic tools
6. **Develop** a personal policy for responsible AI-assisted coding

## Timing

TODO: this needs actual timing

Estimated total time: XXX hours

| Section | Time | Notes |
|---|---|---|
| Introduction to LLMs | 30 min | Can be shortened for technical audiences |
| Scenario I: Full control | 30 min | Core concepts, many exercises |
| Scenario II: IDE integration | 30 min | Hands-on setup possible |
| Scenario III: Agentic | 30 min | Discussion-heavy |
| Security | 30 min | Critical section, don't rush |
| Conclusion | 15 min | Reflection and planning |

**For a 2-hour workshop**: Skip some exercises, focus on Scenarios I and II, and the security section.

**For a half-day workshop**: Include all exercises and add hands-on setup time.

## Preparing exercises

### Before the workshop

1. **Check institutional policies**: Some institutions restrict AI tool use. Know the local rules before teaching or make sure learners are aware of their university's rules..
2. **Test tool access**: Verify that learners can access at least one AI chatbot. We recommend Duck.ai as it requires no account and is privacy-focused. Alternatives include ChatGPT/Copilot, Claude, or Gemini. Some Universities might provide some local implementation of these tools, and those can also be used.
3. **Prepare fallback examples**: If learners can't install (and don't want to install) IDE extensions or agents, have screenshots or recordings ready.
4. ??? DO WE NEED THIS ??? **Create a test repository**: Set up a simple Python project for sandbox exercises if using Docker.

### Exercise preparation

- **Chat exercises (Scenario I)**: Work with any AI chatbot, no special setup
- **IDE exercises (Scenario II)**: Requires VS Code and extension installation (may take 10-15 minutes for setup + accounts)
- **Security exercises**: Can be done as discussion if time is short
- **Sandboxing exercises**: Require Docker (optional, can demonstrate instead or leave it for self-learning)

## Other practical aspects

### Audience considerations

**For researchers/scientists:**

- Emphasize data sensitivity and reproducibility
- Connect to research integrity discussions
- Discuss publication and funding agency requirements

**For developers/RSEs:**

- Can move faster through basics
- Emphasize security and code review practices
- Discuss team workflow integration

**For mixed audiences:**

- Start with common ground (everyone uses or will use these tools)
- Use research examples but acknowledge varied backgrounds

### Tools needed

- Learners need laptops with:
  - Web browser (for chatbots)
  - VS Code (for IDE integration section)
  - Docker (optional, for sandboxing)

### Handling sensitive topics

- **Privacy concerns**: Acknowledge that data handling varies by tool and tier. Don't dismiss concerns.
- **Job security fears**: Frame AI as a tool that augments, not replaces. Emphasize the importance of human judgment.
- **Institutional restrictions**: Some learners may not be allowed to use certain tools. Provide alternatives (local models, chat-only approaches).
- **Sustainability**: Sustainability it is not only about the environment, but also the fact that the most used tools are controlled by few private entities and their goal is to maximise their profits (more tokens might mean better performance, but also more revenue). Long-term use of such tool might erode critical thinking capabilities of the user.

## Setting realistic expectations

One of the most important things instructors can do is set realistic expectations about what learners will experience.

### LLMs amplify existing expertise

Experienced developers get dramatically better results from AI coding assistants than beginners (TODO: enrico to add the citation). This is because they:

- Know what to ask for (have a mental model of the solution)
- Can evaluate whether output is correct
- Know which follow-up questions to ask
- Recognize when the AI is confidently wrong
- Understand the ecosystem (libraries, patterns, best practices)

**Instructor note**: Be explicit that a learner with basic Python skills won't get the same results as someone with years of experience. This isn't a failure of the tools or the learner—it's the nature of amplification.

### Frame the benefit correctly

The real benefit isn't "coding faster"—it's **enabling projects that wouldn't otherwise exist**. For researchers, this might mean:

- Building a quick visualization they wouldn't have time to code manually
- Prototyping approaches before committing
- Creating small utilities that improve workflow

Use this framing to avoid disappointment when AI doesn't instantly solve complex problems.

### The testing imperative

One thing that cannot be outsourced: **verifying that code actually works**.

> *"Your responsibility as a software developer is to deliver working systems. If you haven't seen it run, it's not a working system. You need to invest in strengthening those manual QA habits."* — Simon Willison

Emphasize this repeatedly. The most common failure mode is accepting code that looks right but hasn't been tested.

## Interesting questions you might get

**"Is AI-generated code plagiarism?"**

- As usually, it depends (e.g. on your institution's policies and the context)
- Disclosure is generally recommended (see ALLEA principles for researchers in EU)
- Check your journal's/funder's requirements
- The code itself is typically not copyrighted, but be aware of licensing issues with training data

**"What about code completion I've been using for years (IntelliSense, etc.)?"**

- Traditional autocomplete uses static analysis, not LLMs
- It doesn't send your code to external servers
- Modern AI completion is fundamentally different in capability and risk

**"Can I use AI for my thesis/dissertation code?"**

- Check with your supervisor and institution
- Disclose AI use in your methods section
- Be prepared to explain and defend all code
- Document your verification process

**"Which tool is best?"**

- Depends on your use case, risk tolerance, and institutional policies
- There's no single "best" tool AND the landscape changes very fast
- We recommend starting with chat-based approaches, take notes of what worked well for you, improvise and adapt to changes.

**“What about local/offline models?”**

- There are options (Ollama, LM Studio, etc.)
- Quality gap has narrowed significantly (Qwen2.5-Coder rivals GPT-4 on benchmarks **CITATION NEEDED**)
- Good option for sensitive work and offline use
- See appendix-local-llms for setup instructions

## Typical pitfalls

### Teaching pitfalls

1. **Tool-specific focus**: Don’t spend too much time on one tool’s UI. Focus on principles that transfer.
2. **Overselling or underselling**: Neither “AI will replace programmers” nor “AI is useless” is accurate. Stay balanced.
3. **Skipping security**: Security section is often rushed. Don’t skip it, it’s the most important practical content.
4. **Assuming access**: Not everyone can sign up for or afford AI tools. Have free alternatives ready.
5. **Ethicality of technology**: only a handful of AI models have been trained according to pricinples of responsible AI (e.g. SSAFE-D framework: Sustainability, Safety, Accountability, Fairness, Explainability, and Data-Stewardship). Make informed decisions based on your ethical principles.

### Learner pitfalls

1. **Uncritical acceptance**: Learners may start accepting all suggestions without review. Emphasize verification repeatedly.
2. **Over-reliance**: Some learners want AI to do everything. Stress that understanding code is still essential.
3. **Privacy complacency**: “I have nothing to hide” attitude toward data sharing. Use concrete examples of sensitive data.
4. **Skipping verification**: Learners may skip package verification steps. Make this a graded exercise if possible.

## Adapting the material

TODO: this needs rewrite.

### For shorter workshops (1 hour)

Focus on:

- Brief intro (10 min)
- Scenario I with one exercise (20 min)

- Security highlights (20 min)
- Conclusion and resources (10 min)

### For longer courses (half day or more)

Add:

- Hands-on IDE setup with troubleshooting time
- Docker sandbox setup and testing
- Group discussion on institutional policies
- Peer code review of AI-generated code

### For online delivery

- Use breakout rooms for exercises
- Have a shared document for collecting questions
- Consider asynchronous components (pre-reading, post-workshop exercises)
- Test screen sharing of IDE and terminal beforehand

## Resources for instructors

### Training and pedagogy

- CodeRefinery instructor training
- The Carpentries instructor training

### Practitioner perspectives (valuable for examples and discussion)

- Simon Willison: How I use LLMs to help me write code - Detailed practical workflow from an experienced developer
- Simon Willison's AI-assisted programming series - Ongoing collection of posts
- Addy Osmani: My LLM coding workflow - Best practices from a software engineering lead

### Security

- OpenSSF Security Guide - Good source for security examples

### Note on using practitioners' quotes

The course materials include quotes from practitioners like Simon Willison. These serve several purposes:

- **Credibility**: Showing that experienced developers use these tools thoughtfully
- **Nuance**: Practitioners often capture subtleties better than abstract principles
- **Accessibility**: Conversational language can be more memorable than formal statements

When teaching, you can expand on these quotes with your own experience or invite learners to discuss whether they agree with the perspectives shared.

## Who is the course for?

This course is designed for:

- **Researchers at all levels** who write code for data analysis, simulations, or scientific computing and wish to accelerate their coding workflows
- **Research software engineers** who support scientific projects
- **Anyone** curious about using AI assistants responsibly in a research context

No prior experience with AI coding tools is required.

## About the course

This course takes a research-integrity-focused approach on AI-assisted coding. **There is no point in trying to optimise deep knowledge for any of these tools** since what works well today, will most likely be replaced by something else in a few months. Rather than simply teaching you how to use these tools, we help you understand:

- **What happens under the hood**: How do these tools work? What data were they trained on?
- **What leaves your machine**: When you use these tools, what information is sent to remote servers?
- **What risks exist**: From hallucinated packages to prompt injection attacks
- **How to mitigate risks**: Sandboxing, verification strategies, and best practices

## See also

- OpenSSF Security Guide for AI Code Assistants
- TODO Add more references here

## Credits

This lesson was developed as part of CodeRefinery training activities. You can check the list of contributors, submit an issue, or even suggest an edit via pull request, by visiting ADD GITHUB URL HERE.