Good Practices for AI-Assisted Coding

Open methodologies for **N**eurorehabilitation and **D**ata **A**nalysis

Giorgio Arcara

version 0.1

June 1, 2026

This document was drafted in collaboration with an AI language model (Claude, Anthropic). The ideas, structure, and editorial judgement are the author's own; the AI assisted with drafting, phrasing, and formatting.

Contents

1 Introduction **3**

2 The Core Problem **4**

3 Define the Problem First **5**

4 Practices for Experienced Programmers **6**
4.1 Trace before you accept 6
4.2 Design the algorithm first 6
4.3 Periodically code without LLM assistance 7
4.4 Seek friction deliberately 7

5 Practices for Junior Researchers **7**
5.1 Blank-page reconstruction 7
5.2 Inverted rubber duck 7
5.3 Write tests before asking for code 8
5.4 Keep a bug journal 8
5.5 Change one variable at a time when debugging 8
5.6 Decompose the problem yourself before prompting 8

6 Using LLM and privacy **9**

7 Quick Reference **9**

"How we spend our days is, of course, how we spend our lives."
— Annie Dillard, "The Writing Life"

"In every animal... a more frequent and continuous use of any organ gradually strengthens, develops and enlarges that organ... while the permanent disuse of any organ imperceptibly weakens and deteriorates it..."
— Carl Lamarck, "Zoological Philosophy"

Aldo, Giovanni e Giacomo, "Chiedimi se sono Felice"

1 Introduction

Large Language Models (LLMs) such as ChatGPT, Claude, Copilot, and similar tools have become widely used aids in everyday coding tasks. They are undeniably powerful: they can accelerate prototyping, suggest solutions to common problems, explain unfamiliar syntax, and reduce the time spent on boilerplate code.

In a research context, however, this efficiency comes with a risk that is easy to underestimate.

Your aim is framing correctly the problem

In many professional contexts, code is the deliverable: if it works correctly and on time, the job is done. Research is different. Code in research is a *means*, not an end. The real deliverable is framing and solving correctly a given problem, and the solution can only be as solid as the researcher's understanding of the pipeline that produced it.

Consider what it actually means to *not* have a full understanding of your own analysis code:

- You cannot defend your methodological choices, because you did not make them—you accepted them.
- You cannot catch errors before they become published mistakes. A bug that produces plausible-looking output is invisible to someone who does not know what the output *should* look like.
- You cannot adapt the analysis to a new dataset, a reviewer's request, or a follow-up question, because you do not understand the logic well enough to modify it safely.
- You cannot teach or explain your methods to others, which undermines the very transparency that open science requires.

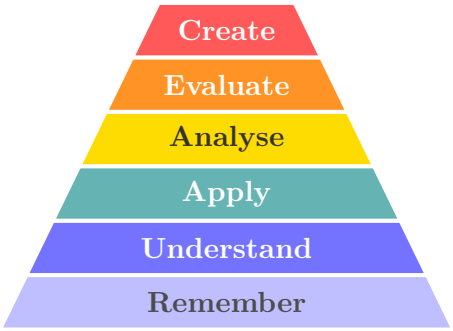
Working code produced without understanding is not a research output. It is a liability.

The understanding you build while writing the analysis is part of the scientific contribution itself, not a side effect of it. Velocity optimised at the cost of that understanding is a net loss, regardless of how clean the final script looks.

This guide does not argue against using LLMs. It argues for using them *deliberately*.

2 The Core Problem

Learning to code—and to solve problems computationally—is a cognitive process that spans multiple levels of competence. Bloom's Taxonomy, a well-established framework in educational psychology, describes six levels of cognitive skill, ordered from the most superficial to the most demanding:



Bloom's Taxonomy of cognitive skills (bottom = most superficial, top = most demanding)

Genuine coding competence sits at the top: **Create**—the ability to design and produce solutions to novel problems. Reaching it requires building through the intermediate levels: understanding why code behaves as it does, applying principles in unfamiliar contexts, analysing trade-offs, evaluating correctness.

LLMs disrupt this ladder. You don't create any working solution even if it may seem so. When a working solution is handed to you by LLM before you have wrestled with the problem, you interact with code at its lowest level—**Remember**: you recognise patterns, run scripts, adjust parameters,

and produce outputs. This is not competence. It produces the illusion of competence.

The risk differs depending on experience level:

- **Experienced programmers** risk the gradual atrophy of higher-order skills—analysis, evaluation, design. The capacity to Create does not disappear overnight, but it quietly erodes as cognitive work is increasingly outsourced.
- **Junior researchers** risk becoming permanently anchored at Remember. Without ever climbing the intermediate levels of the taxonomy, there is no foundation on which genuine competence can develop. Familiarity with a codebase—running scripts, reading outputs—is not understanding it.

Key Principle

All good practices described in this guide share one goal: **pushing you up Bloom's Taxonomy**—away from the passive familiarity that LLMs enable, and toward the active competence that research demands.

3 Define the Problem First

Before any of the practices in the following sections can help, there is a prerequisite step that applies to everyone, regardless of experience level: **define the problem before you open the editor—or the LLM.**

In research, coding exists to solve a problem. That sounds obvious, but it has a non-obvious implication: if the problem is not clearly defined, no amount of working code solves it. An LLM is very good at solving the problem you give it—the risk is giving it the wrong problem because you have not thought it through yourself first.

Defining the problem means being able to answer, in plain language, *before* writing a single line:

- What question am I trying to answer?
- What does the input look like, and what should the output look like?

- What would a correct result look like, and how would I recognise a wrong one?

If you cannot answer these questions, you are not ready to code—and you are certainly not ready to prompt. This step is not a practice to add to a checklist: it is the foundation that makes every other practice meaningful.

4 Practices for Experienced Programmers

If you already have solid programming experience, the primary risk is not that you will fail to understand LLM output—it is that you will stop *bothering* to. The practices below are designed to keep your problem-solving muscle active.

4.1 Trace before you accept

Before accepting any non-trivial block of LLM-generated code, trace through it mentally (or with pen and paper). Ask yourself:

- What does each step do?
- What are the edge cases or failure modes?
- Would I have written it differently, and why?

If you cannot answer these questions, do not commit the code.

4.2 Design the algorithm first

For any problem that requires more than a few lines of logic, sketch the algorithm yourself before prompting. Use pseudocode, a diagram, or plain English. Then use the LLM to *implement* your design, not to invent it.

Pattern to avoid: *"Here is my data. Write me a function that does X."*

Pattern to prefer: *"I want to loop over rows, filter by condition Y, and aggregate using Z. Help me implement this."*

4.3 Periodically code without LLM assistance

Schedule regular sessions—even short ones—where you work entirely without LLM assistance. These serve as a diagnostic: if a task you once found easy now feels surprisingly difficult, that is a signal worth taking seriously.

4.4 Seek friction deliberately

Efficiency is not always the goal. When learning a new library, domain, or language, resist the temptation to let the LLM smooth every bump. Struggle is the mechanism of learning.

5 Practices for Junior Researchers

If you are early in your programming journey, the risk is subtler and more insidious: you can develop *operational fluency*—the ability to run, modify, and talk about code—without a corresponding mental model. The practices below are self-directed and do not require a mentor.

5.1 Blank-page reconstruction

After receiving and using a piece of LLM-generated code, close it. Then try to rewrite it from scratch without looking. Do not aim for perfection—aim for understanding. Where you get stuck reveals exactly what you have not yet understood.

5.2 Inverted rubber duck

The classical "rubber duck debugging" technique involves explaining your code out loud to force clarity. The *inverted* version: explain the LLM's code

back to the LLM in your own words, then ask it to identify any gaps or errors in your explanation.

Example prompt

"Here is my understanding of what this function does: [your explanation]. Are there any steps I have described incorrectly or any important aspects I have missed?"

5.3 Write tests before asking for code

Before asking the LLM to write a function, write down (in plain language or actual test cases) what correct behaviour looks like: given this input, what should the output be? This forces you to understand the problem before you receive the solution.

5.4 Keep a bug journal

Keep a running log of bugs you have encountered and how they were resolved. Focus on the *why*: what was your incorrect mental model, and how did it get corrected? Over time, this journal becomes a record of your understanding growing.

5.5 Change one variable at a time when debugging

When something does not work, resist the temptation to ask the LLM to fix it immediately. Debugging is precisely where you practise the higher levels of Bloom's Taxonomy—Analyse and Evaluate—and those levels cannot be climbed by delegation. Form a hypothesis about what is wrong, change exactly one thing, and observe the result. Only reach for the LLM after you have made at least one genuine attempt to diagnose the problem yourself.

5.6 Decompose the problem yourself before prompting

For any non-trivial task, write down the sub-problems before opening the LLM. This does not need to be formal—a rough bullet list is enough. The act of decomposition is where much of the learning happens, and it is precisely what gets skipped when you prompt immediately.

6 Using LLM and privacy

Using LLM for data analysis is at risk of *data breach*, and this should not be taken as a irrelevant.

Key Principle

Never, ever, upload your data to an LLM to do the analysis or to help you solving a problem.

If you do this, please consider you just commited a *data breach*.

7 Quick Reference

Everyone: define the problem before coding or prompting	
Experienced programmers	**Junior researchers**
Trace all LLM code before accepting	Blank-page reconstruction after using LLM code
Design the algorithm before prompting	Inverted rubber duck (explain code back, ask for gaps)
Code without LLM regularly (diagnostic)	Write test cases before asking for code
Seek friction deliberately when learning	Keep a bug journal (focus on the *why*)
	Change one variable at a time when debugging
	Decompose the problem yourself before prompting

Shared principle for both groups: *Use the LLM to implement and to check, but not to think in your place.* The goal is not to avoid LLMs. The goal is to remain the author of your own understanding.