Good Practices for AI-assisted coding

Giorgio Arcara

05/06/2026

Aphorism 1

"How we spend our days is, of course, how we spend our lives."

Annie Dillard, "The Writing Life' '

Aphorism 2

"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' '

Aphorism 3


CHI SA FARE, SA CAPIRE
MA CHE CAZZO DI PROVERBIO È?
NON È UN PROVERBIO, GIACOMO, È LA VITA

LLMs are powerful tools...

- They accelerate prototyping
- They explain unfamiliar syntax and suggest solutions
- They lower the barrier to getting something working

LLMs are powerful tools...

- They accelerate prototyping
- They explain unfamiliar syntax and suggest solutions
- They lower the barrier to getting something working

...but in research, *working code* is not the goal.

Solving the problem is the aim.

In research, code is a **means**, not an end. The real deliverable is a **solving a specific research problem** — and that claim can only be as solid as the researcher's understanding of the pipeline that produced it.

Solving the problem is the aim.

In research, code is a **means**, not an end. The real deliverable is a **solving a specific research problem** — and that claim can only be as solid as the researcher's understanding of the pipeline that produced it.

Not having a mental model means:

- You cannot **defend** your methodological choices
- You cannot **catch errors** before they become published mistakes
- You cannot **adapt** the analysis to a new dataset or reviewer request
- You cannot **teach** your methods to others

Solving the problem is the aim.

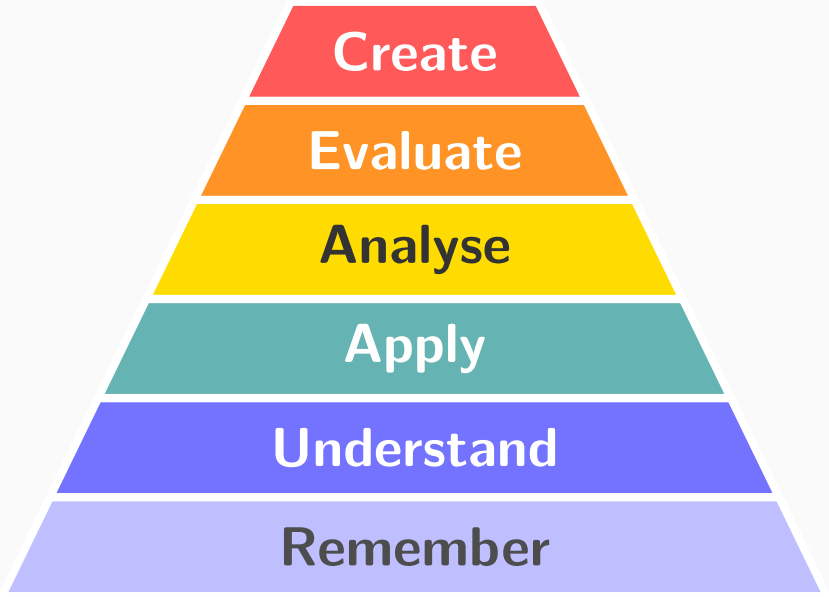
In research, code is a **means**, not an end. The real deliverable is a **solving a specific research problem** — and that claim can only be as solid as the researcher's understanding of the pipeline that produced it.

Not having a mental model means:

- You cannot **defend** your methodological choices
- You cannot **catch errors** before they become published mistakes
- You cannot **adapt** the analysis to a new dataset or reviewer request
- You cannot **teach** your methods to others

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

Bloom's Taxonomy of cognitive skills



The core problem

Bloom's Taxonomy spans from the most superficial level (**Remember**) to the most demanding (**Create**).

Genuine coding competence sits at the top: **Create** — the ability to design and produce solutions.

The core problem

Bloom's Taxonomy spans from the most superficial level (**Remember**) to the most demanding (**Create**).

Genuine coding competence sits at the top: **Create** — the ability to design and produce solutions.

LLMs anchor you at the bottom.

When a working solution is handed to you before you have wrestled with the problem, you interact with code at its lowest level — **Remember**: recognising patterns, running scripts, adjusting parameters.

The core problem

Bloom's Taxonomy spans from the most superficial level (**Remember**) to the most demanding (**Create**).

Genuine coding competence sits at the top: **Create** — the ability to design and produce solutions.

LLMs anchor you at the bottom.

When a working solution is handed to you before you have wrestled with the problem, you interact with code at its lowest level — **Remember**: recognising patterns, running scripts, adjusting parameters.

> All good practices in this guide share one goal: **pushing you up Bloom's Taxonomy** — away from familiarity, toward genuine competence.

Who is at risk?

Experienced programmers

Risk: gradual *atrophy* of problem-decomposition skills. The skills do not disappear overnight — they quietly erode as cognitive work is increasingly outsourced.

Junior researchers

Risk: becoming anchored at *Remember* — mistaking familiarity for understanding. It is possible to run scripts, adjust parameters, and produce outputs without any mental model of what the code does. Research demands *Create*.

Before any practice: define the problem

In research, coding exists to **solve a problem**. This 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.

Before any practice: define the problem

In research, coding exists to **solve a problem**. This 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.

Before writing a single line of code, be able to answer in plain language:

- 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?

Before any practice: define the problem

In research, coding exists to **solve a problem**. This 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.

Before writing a single line of code, be able to answer in plain language:

- 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 not ready to prompt.

Practices for Experienced Programmers

Practices for experienced programmers

Trace before you accept
Before committing any LLM-generated code, trace through it mentally. Ask: *What does each step do? What are the failure modes?*

Design the algorithm first
Sketch the algorithm yourself *before* prompting. Use the LLM to *implement* your design — not to invent it.

Code without LLM regularly
Schedule sessions without LLM assistance as a diagnostic. If a familiar task feels surprisingly hard, that is a signal.

Seek friction deliberately
When learning something new, resist letting the LLM smooth every difficulty. Struggle is the mechanism of learning.

Practices for Junior Researchers

Practices for junior researchers (1)

Blank-page reconstruction
After using LLM code, close it and rewrite from scratch. Where you get stuck reveals exactly what you have not understood.

Inverted rubber duck
Explain the LLM's code back to it in your own words, then ask it to find gaps in your explanation.

Write tests before asking for code
Write down what correct behaviour looks like *before* asking the LLM. This forces you to understand the problem first.

Practices for junior researchers (2)

Keep a bug journal

Log bugs and how they were resolved. Focus on the *why*: what was your incorrect mental model, and how did it get corrected?

Change one variable at a time

Debugging is where you practise Analyse and Evaluate — the levels that cannot be climbed by delegation. Form a hypothesis, change exactly one thing, and observe. Only reach for the LLM after at least one genuine attempt yourself.

Decompose before prompting

For any non-trivial task, write the sub-problems down first. The act of decomposition is where much of the learning happens.

Summary

Everyone: define the problem before coding or prompting.

Experienced programmers

- Trace LLM code before accepting
- Design algorithm before prompting
- Code without LLM (diagnostic)
- Seek friction deliberately

Junior researchers

- Blank-page reconstruction
- Inverted rubber duck
- Tests before code
- Bug journal
- One variable at a time
- Decompose before prompting

Use the LLM to implement and to check — not to think in your place.
The goal is to remain the *author of your own understanding*.

But ...

What are we *losing*, in the long run?

See here