PROMPTING FOR PROGRAMMERS

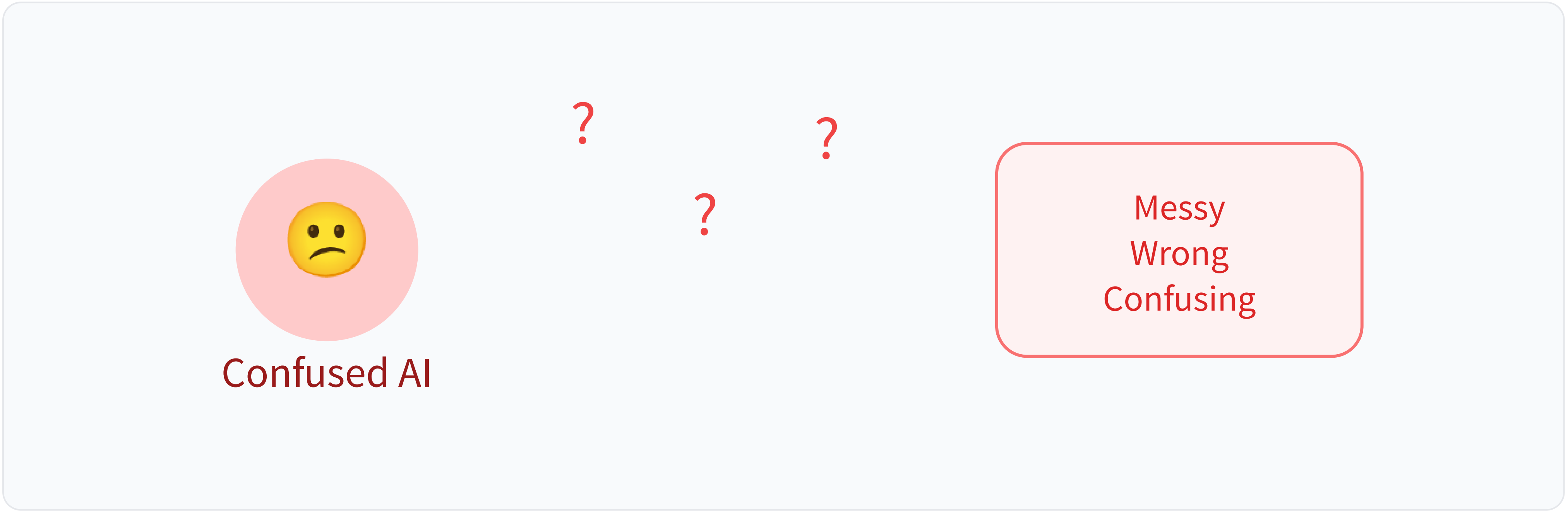
FROM VAGUE ASKS TO REPRODUCIBLE RESULTS

WHAT YOU'LL LEARN TODAY

- A mental model for **how prompting works**
- A simple recipe for **writing good prompts**
- How to ask **clarifying questions** before coding
- Setting clear **constraints and non-goals**
- Advanced techniques like **Persona Pattern** and **Chain-of-Thought**
- Professional approaches: **Tests-first**, **patch/diff style**, and **repo context**
- Understanding **instruction hierarchy** and **AI limitations**

BAD WAY TO ASK AI FOR HELP

> "WRITE A PYTHON FUNCTION"



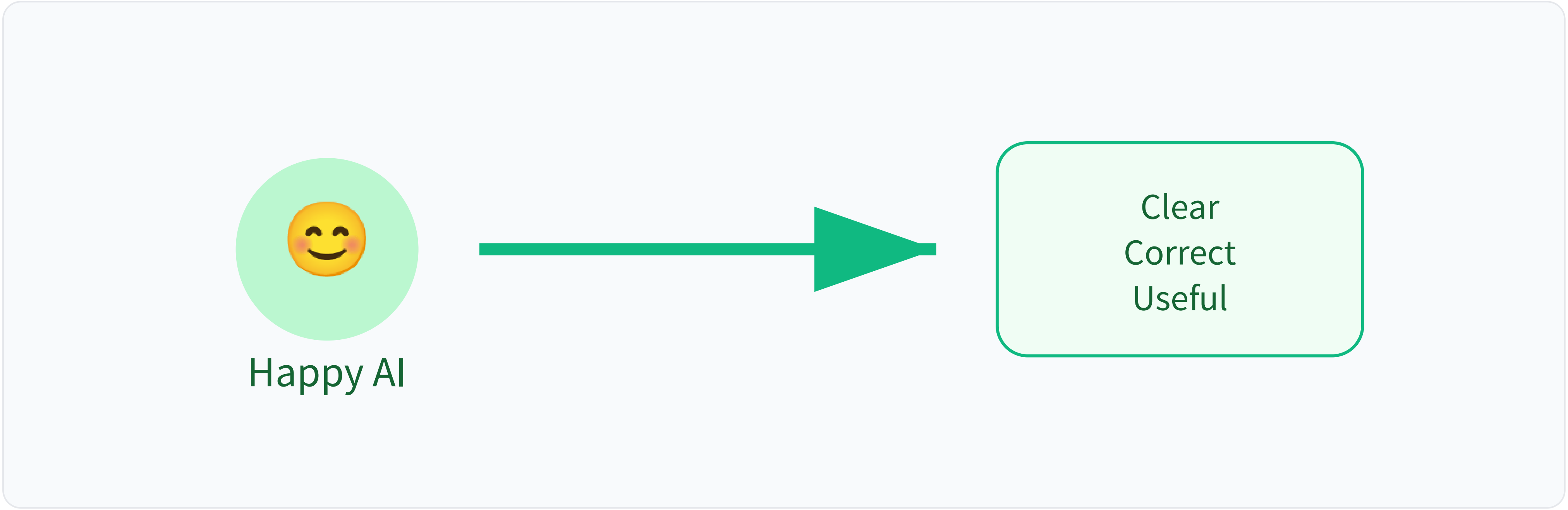
WHY THIS IS BAD:

- AI doesn't know what you really want

- You get random, unhelpful code
- Takes forever to fix

GOOD WAY TO ASK AI FOR HELP

```
"Write a Python function that adds two numbers.
Call it 'add_numbers'.
It should take two numbers and return their sum.
Include a simple example of how to use it."
```



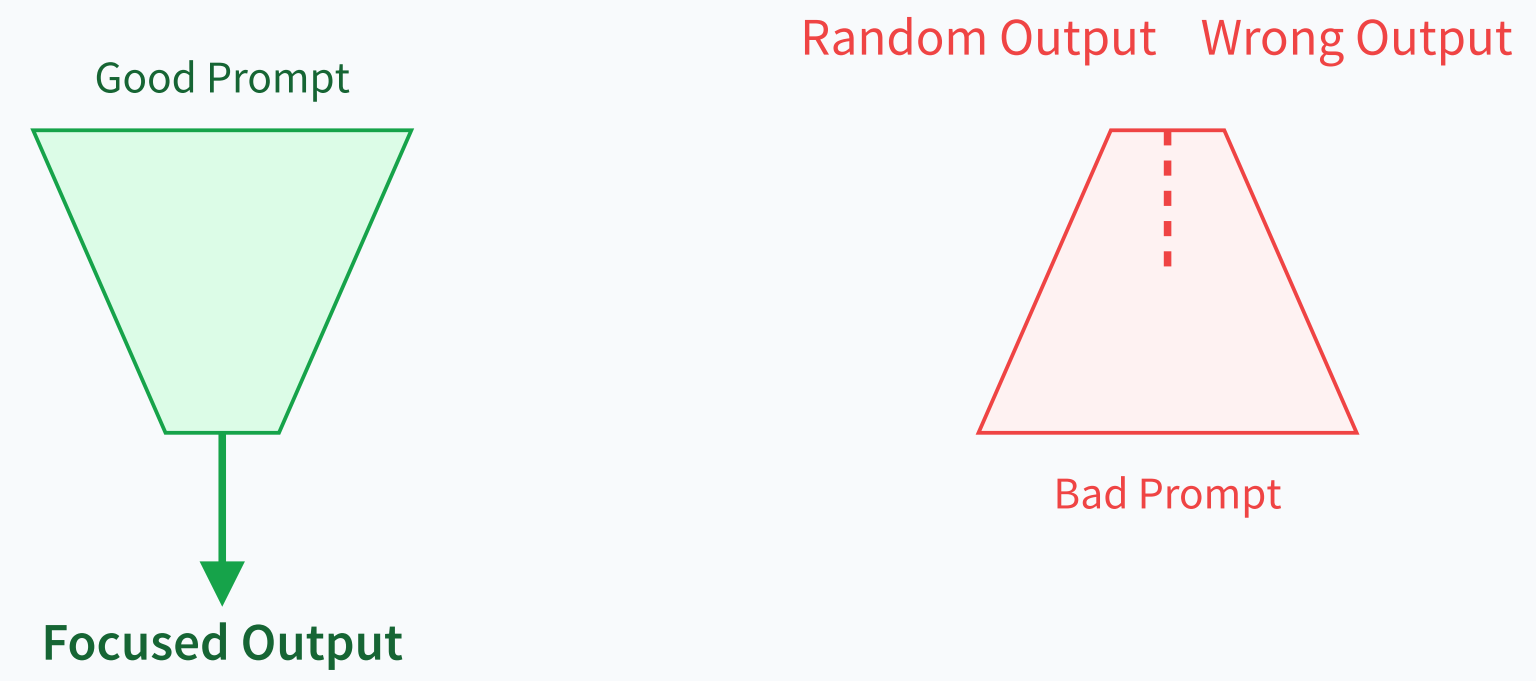
WHY THIS WORKS BETTER:

- AI knows exactly what you want
- You get helpful, working code
- Saves you time!

MENTAL MODEL: FOCUSING THE AI

- AI generates code based on **patterns** it has seen
- A vague prompt gives it **too many possibilities**
- A good prompt **narrows the possibilities** to what you want
- This reduces errors and "hallucinations"

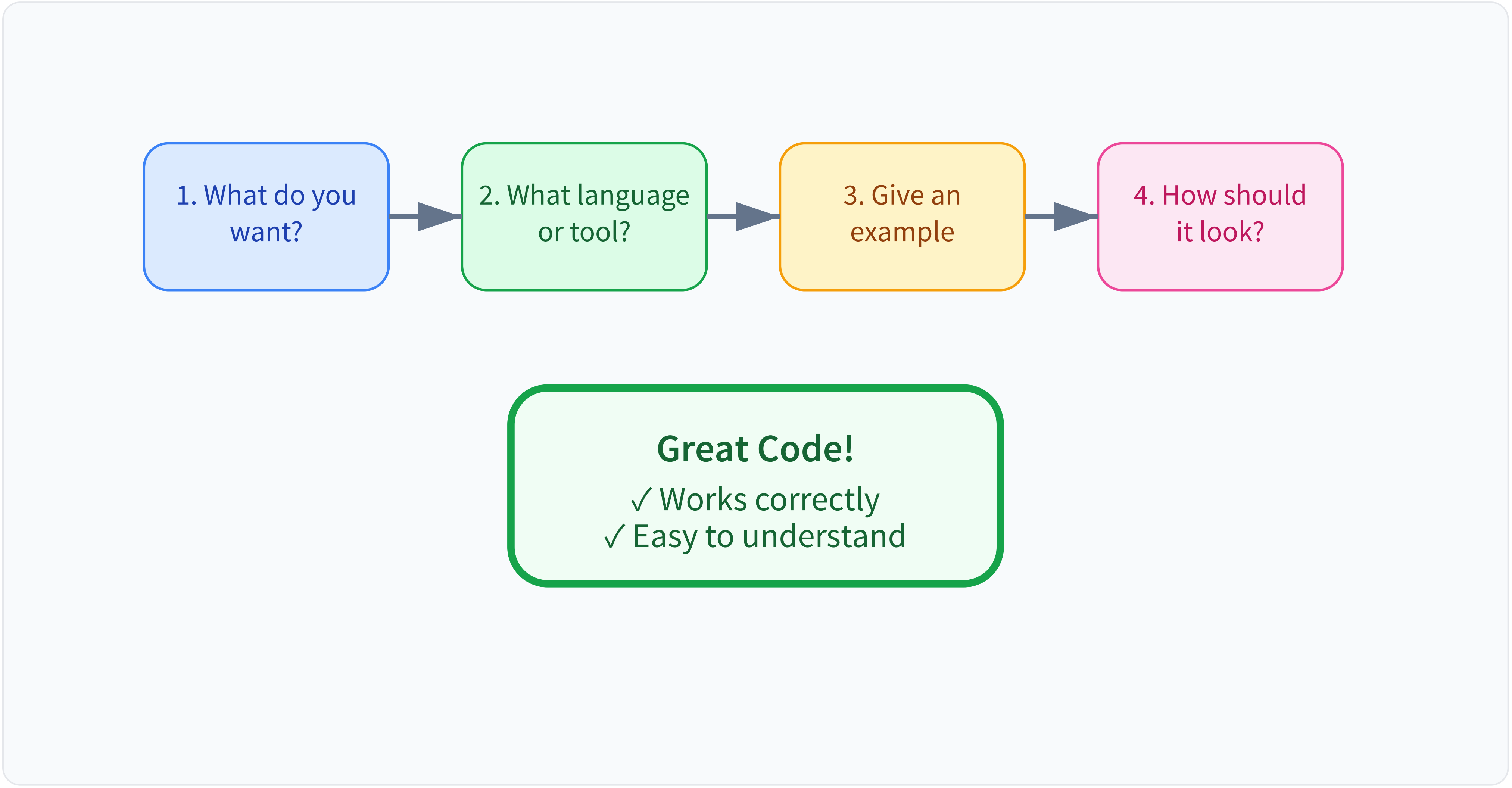
PROMPT QUALITY DETERMINES OUTPUT FOCUS



A clear prompt acts like a funnel, guiding the AI to the correct result.

Think of it like giving directions: "Go to the city" vs. "Go to 221B Baker Street, London".

SIMPLE RECIPE FOR GOOD PROMPTS



Follow these 4 steps and you'll get much better help from AI!

LET'S SEE THIS RECIPE IN ACTION

TASK: CREATE A SIMPLE CALCULATOR

❌ BAD PROMPT:

```
"Make a calculator"
```

✅ GOOD PROMPT:

```
"Create a Python function called 'calculate'
that can add, subtract, multiply, and divide
two numbers.

For example: calculate(5, 3, '+') should return 8

Make it return the result as a number."
```

See how the good prompt follows our 4-step recipe?

LET'S PRACTICE TOGETHER!

YOUR TURN: WRITE A GOOD PROMPT

You want AI to help you create a program that asks someone their name and says hello to them.

THINK ABOUT OUR 4 STEPS:

1. What do you want? (A greeting program)
2. What language? (Python)
3. Give an example (Input: "Alice", Output: "Hello Alice!")
4. How should it look? (Simple and easy to read)

Take 2 minutes: Write your prompt with a partner!

HERE'S ONE GOOD EXAMPLE

```
"Write a Python program that asks the user to type their name,
then prints a friendly greeting.

For example:
- If the user types 'Alice', it should print 'Hello Alice!'
- If the user types 'Bob', it should print 'Hello Bob!'

Make the code simple and add comments to explain what it does."
```

WHY THIS WORKS:

- ✅ Clear goal (greeting program)
- ✅ Specific language (Python)
- ✅ Good examples (Alice, Bob)
- ✅ Clear format (simple + comments)

CONSTRAINTS: WHAT YOU MUST FOLLOW

Remember our **4-step recipe**? After clarifying what you want, specify the **rules and boundaries** AI must respect.

WHY CONSTRAINTS MATTER:

Clear boundaries help AI focus its suggestions within your project's requirements. Think of it as giving AI the "rules of the game."

✅ COMMON CONSTRAINT CATEGORIES

- **Tech Stack:** Python 3.11, Node 20, React 18
- **Code Style:** ESLint rules, Prettier formatting
- **Performance:** Under 200ms response time
- **Security:** No eval(), validate all inputs
- **Compatibility:** No breaking changes to API
- **Testing:** Must include unit tests

EXAMPLE WITH CONSTRAINTS:

```
**Task:** Add email validation to user registration
**Constraints:**
- Use existing Joi validation library
- Return 400 status with clear error message
- Must work with current Express middleware
- Follow existing error handling pattern
```

NON-GOALS: WHAT YOU SHOULD AVOID

Just as important as saying what to do: **explicitly state what NOT to do**. This prevents scope creep and unwanted changes.

WHY NON-GOALS MATTER:

AI might suggest "helpful" extras that break your system. Non-goals act like a fence to keep solutions focused and safe.

❌ COMMON NON-GOAL CATEGORIES

- **No DB changes:** Keep existing schema
- **No new dependencies:** Use current libraries
- **No framework upgrades:** Stay on current version
- **No UI changes:** Backend-only modifications
- **No major refactors:** Minimal, focused changes
- **No auth changes:** Keep existing security model

EXAMPLE WITH NON-GOALS:

```
**Task:** Add email validation to user registration
**Non-Goals:**
- Don't modify the database schema
- Don't change frontend validation logic
- Don't add new npm dependencies
- Don't alter the existing user model
```

POPULAR PROMPTING FRAMEWORKS

Professional developers use these memorable acronyms:

⭐ STAR METHOD

- Situation: Context and background
- Task: What you want to accomplish
- Action: Specific steps to take
- Result: Expected outcome format

🎯 CLEAR FRAMEWORK

- Context: Provide background info
- Length: Specify output length
- Examples: Give sample inputs/outputs
- Audience: Who will use this?
- Role: What expert should AI be?

🚀 CREATE METHOD

- Character: AI's role/persona
- Request: Clear task description
- Examples: Sample inputs/outputs

🏗️ SPEC (OUR FRAMEWORK)

- Specific goal: What you want
- Programming language/tool
- Example: Sample input/output

- **A**djustments: Refinements needed
- **T**ype: Format of response
- **E**xtras: Additional requirements

- **C**onstraints: How it should look

Pro tip: Pick one framework and stick with it to build consistency!

CLARIFYING QUESTIONS

Ask before you code when goals or constraints are ambiguous.

USEFUL STEMS

- "What are the **acceptance criteria** for this change?"
- "Which **interfaces or files** must stay backward compatible?"
- "Any **non-goals** I should explicitly avoid?"
- "What **deadline** and **scope** do we have?"
- "Should I prefer a **minimal diff** or a refactor?"

QUICK TEMPLATE

```
Before I start, a couple of quick checks:
- Goal and success criteria?
- Constraints (APIs, style, frameworks)?
- Non-goals / out of scope?
```

```
  - Preferred output (diff, file, snippet)?
  - Any tests, data, or secrets to use/avoid?
```

THE PERSONA PATTERN

Tell the AI to act as an expert with a specific role.

❌ GENERIC PROMPT

```
"Review my Python code for errors."
```

AI gives: "Looks okay."

❌ Basic, unhelpful feedback.

✅ WITH PERSONA PATTERN

```
"Act as a senior Python developer and a security expert.
Review my Python code.
Look for subtle bugs, performance issues, and security vulnerabilities.
Explain your findings with code examples."
```

AI gives: "Found a potential SQL injection vulnerability..."

✅ Expert-level, actionable advice!

Why it works: You focus the AI on a specific knowledge set, unlocking more detailed and relevant insights.

CHAIN-OF-THOUGHT PROMPTING

Make AI show its "thinking" process step-by-step

❌ WITHOUT CHAIN-OF-THOUGHT

```
"Solve this Python problem:
Find the second largest number in [3, 1, 4, 1, 5, 9]"
```

AI might give: "The answer is 5"

❌ No explanation, hard to verify, might be wrong

✅ WITH CHAIN-OF-THOUGHT

```
"Solve this Python problem step by step:
Find the second largest number in [3, 1, 4, 1, 5, 9]

Think through it:
1. First, what's the process?
2. Show your work
3. Then give the final answer"
```

AI gives: "1. Remove duplicates: [3,1,4,5,9]
2. Sort: [1,3,4,5,9]
3. Second largest: 5"

✅ Clear reasoning, easy to check!

Magic phrases: "Think step by step", "Show your work", "Explain your reasoning"

FEW-SHOT VS. ZERO-SHOT

Giving the AI examples vs. no examples

❌ ZERO-SHOT (NO EXAMPLES)

```
"Convert 'apple' to pig latin."
```

AI might give: "Appleay"

❌ Correct, but maybe not the format you want.

✅ FEW-SHOT (WITH EXAMPLES)

```
"Convert words to pig latin.
'banana' -> 'ananabay'
'hello' -> 'ellohay'
'apple' -> ?"
```

AI gives: "appleay"

✅ Follows your exact format!

Pro tip: Use few-shot prompting when you need a very specific output format or style.

INSTRUCTION HIERARCHY

Rule of Thumb When instructions conflict, follow the highest-priority source.

- **Repo-level guidance** (e.g., `.github/copilot-instructions.md`, path rules) → highest priority
- **File-level constraints** (existing code style, framework conventions)
- **Explicit prompt/task text** (what you ask the model to do)
- **Inline comments** and local context
- **Model defaults** and general knowledge → lowest priority

Example: "For `lectures/**/*.html`, keep Reveal.js structure and `Reveal.initialize` intact." If a prompt asks to overhaul the deck framework, decline or propose a safe alternative.

AI LIMITATIONS & GOTCHAS

What AI **can't do** (yet) - stay alert for these!

🚫 CURRENT LIMITATIONS

- **No real-time data:** Training cutoff dates
- **Can't run/test code:** Logical errors slip through
- **No project context:** Doesn't know your full codebase
- **Security blind spots:** May suggest vulnerable patterns
- **Overconfident:** Sounds sure even when wrong

⚠️ WATCH OUT FOR

- **Hallucinated APIs:** Invents non-existent functions
- **Outdated syntax:** Uses old language versions
- **Copy-paste traps:** Code that "looks right" but isn't
- **Cargo cult programming:** Complex solutions to simple problems
- **Missing edge cases:** Happy path only

✅ YOUR DEFENSE STRATEGY

- **Always test the code** AI gives you
- **Ask for explanations** when something seems complex
- **Cross-check documentation** for API calls
- **Start simple,** then add complexity
- **Remember:** You're still the programmer!

TESTS-FIRST PROMPTING

Write or provide tests first; have the model implement only what’s needed to pass.

EXAMPLE TEST (JS)

```
// email.spec.js
import { isValidEmail } from './email.js'

test('valid emails', () => {
  expect(isValidEmail('a@b.com')).toBe(true)
})

test('invalid emails', () => {
  expect(isValidEmail('not-an-email')).toBe(false)
})
```

PROMPT

```
Implement only the code needed to make these tests pass.
Return a single file:

  email.js

 with a named export

  isValidEmail
```

```
.
No extra commentary.
```

MINIMAL SOLUTION

```
// email.js
export function isValidEmail(s) {
  return /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(s)
}
```

PATCH/DIFF STYLE CHANGES

Ask for unified diffs to keep reviews tight and auditable.

```
diff --git a/utils/math.js b/utils/math.js
index e69de29..4b825dc 100644
--- a/utils/math.js
+++ b/utils/math.js
@@
-export function add(a,b){return a+b}
+export function add(a, b) {
+  if (typeof a !== 'number' || typeof b !== 'number') {
+    throw new TypeError('add expects numbers')
+  }
+  return a + b
+}
```

Tip: For multi-file edits, ask for one diff per file, clearly separated.

REPO CONTEXT & COPILOT INSTRUCTIONS

Give the model concrete paths and rules so outputs align with your project.

INCLUDE CONTEXT

```
Context:
- Follow

  .github/instructions/lectures.instructions.md


- Keep

  Reveal.initialize

 block and slide sizing intact
- Edit only:

  lectures/lecture3-prompting-for-programmers.html



Task:
```

BENEFITS

- Reduces back-and-forth and rework
- Matches repository style and constraints
- Safer, smaller diffs that are easy to review
- Plays nicely with CI and automation

Speaker notes