# MODULE 2 — Human–AI Collaboration & Prompt Engineering for Data Science

**How Module 2 Is Structured (Same as Before)**

Each submodule follows:

1. Concept
2. Concrete example
3. Why this matters for AI-driven data science
4. Mini-exercise

You'll notice this module feels less like algorithms and more like systems engineering + project management, but it's every bit as technical.

---

## ◇ Module 2.1 — AI as a Junior Engineer (Mental Model)

### ☑ Concept

The most important mental model for working with AI in technical projects is this: Treat the AI like a fast junior engineer who: - writes code quickly - needs clear requirements - can misunderstand ambiguity - benefits from plans, constraints, and feedback - should not be trusted blindlyThis model immediately improves:

- code quality
- refactor safety
- iteration speed
- stakeholder confidence

---

### 📌 Example: Bad vs Good Prompting

**❌ Vague prompt (what not to do)**

```
Build me a data pipeline for cleaning a dataset.
```

Likely outcome:

- unclear scope
- arbitrary decisions
- no modularity
- hard to iterate
- brittle code

---

**☑ Structured prompt (what to do)**

```
You are acting as a junior data engineer.

Goal:
Build a data-cleaning pipeline for tabular CSV data.

Constraints:
- Python 3.10
- Modular functions
- No external services
- Deterministic behavior
- Must be easy to refactor

Plan first:
1. Propose a high-level pipeline (steps only, no code)
2. Get approval
3. Implement each step as a separate function
4. Return a minimal working example
```

This mirrors how you'd work with a human teammate.

Q: How does the structure of the prompt scale with more complex data science problems?

A: Yes — for complex data science work, you usually set the goal for the entire envisioned pipeline, but you do not ask the AI to build it all at once.

- In comparison of the simple prompt and the more complex, multi-step prompt there's a few more details added:
  - Long-term & short-term goals are separated out
    - Example: we're going to do X, Y, & Z, but we need to focus on X now
  - Prompt explicitly states not to build out the entire workflow (yet)
- Gives AI foresight (helps AI internalize strategy) without getting lost in the weeds
- Prompts from one sprint to another build off each other

**☑ Good Structured Prompt (End-to-End Goal, Agile Execution)**

Here's the kind of prompt you actually want to use.

```
You are acting as a junior data scientist / data engineer.

End-to-end goal (context only):
We are building a reproducible data science pipeline that will eventually include:
- data ingestion
- schema validation
- data cleaning
- feature engineering
- model training
- model evaluation
```

```
Do NOT implement the full pipeline yet.

Current scope (this sprint):
Focus only on the first two stages:
1) loading tabular CSV data
2) validating basic schema assumptions (column presence, types, missingness)

Constraints:
- Python 3.10
- pandas only (no sklearn yet)
- Modular functions
- Deterministic behavior
- No model training in this step
- Code should be easy to extend later

Plan first:
1) Propose a high-level pipeline outline for the *full* workflow (steps only)
2) Clearly mark which steps are IN SCOPE vs OUT OF SCOPE
3) Wait for approval before writing any code

Deliverable:
- A short written plan
- Then, after approval, Python functions for data loading and schema validation
only
```

**Why this works**

- The AI understands the destination
- It doesn't guess future architecture
- It doesn't overbuild
- You retain control over iteration

---

**How This Evolves Across Iterations (Very Important)**

Next sprint, your follow-up prompt might be:

```
Using the previously approved pipeline structure:

Current scope:
- Add a data cleaning step that handles missing values
- Do not modify existing functions unless necessary

Constraints:
- Backwards compatibility with prior outputs
- New logic must be isolated in its own function

Plan first, then implement.
```

Later:

```
Extend the pipeline to include feature engineering.
Do not touch ingestion or cleaning logic.
```

This is AI-assisted agile development, not "prompt-and-pray."

**Short Answer (the core idea)**

> Yes — for complex data science work, you usually set the goal for the entire envisioned pipeline, but you do not ask the AI to build it all at once.

---

## Key Rule of Thumb (Memorize This)

> **State the full vision once. Execute one slice at a time. Force planning before coding. Never let AI decide scope implicitly.**

If you follow that rule, AI becomes:

- predictable
- refactor-friendly
- stakeholder-safe
- genuinely useful for data science

---

### 💡 Why This Matters for AI-Driven Data Science

AI is very good at:

- **local correctness**
- **pattern completion**
- **rapid implementation**

AI is *not* good at:

- inferring unstated constraints
- long-term project context
- implicit stakeholder requirements
- architectural intent

So the **burden of structure lives with you**.

This module teaches you how to:

- externalize intent
- control scope
- guide planning vs execution
- iterate safely (agile-compatible)

---

Key Rule of Thumb (Memorize This)

State the full vision once. Execute one slice at a time. Force planning before coding. Never let AI decide scope implicitly.If you follow that rule, AI becomes:

- predictable
- refactor-friendly
- stakeholder-safe
- genuinely useful for data science

---

💡 Why This Matters for AI-Driven Data Science

AI is very good at:

- local correctness
- pattern completion
- rapid implementation AI is not good at:
- inferring unstated constraints
- long-term project context
- implicit stakeholder requirements
- architectural intent So the burden of structure lives with you. This module teaches you how to:
- externalize intent
- control scope
- guide planning vs execution
- iterate safely (agile-compatible)

---

## 🧪 Mini-Exercise 2.1

Rewrite this vague prompt into a structured, junior-engineer-style prompt:

```
Create a Python solution to analyze a dataset.
```

**Requirements**

Your rewritten prompt must explicitly include:

- a **role** for the AI
- a **goal**
- at least **3 constraints**
- a **plan-first instruction**
- a clear **deliverable** You don’t need to run any code — just write the prompt.

---

**📤 How to submit**

Reply with:

```
**Original Prompt**
...

**Rewritten Structured Prompt**
...
```

---

## ◇ Module 2.2 — Separating Planning from Execution

> **Theme: Forcing AI to think first and act later**

---

### Why This Module Exists

Left unchecked, AI will:

- jump straight to code
- mix architecture with implementation
- lock in poor assumptions
- make refactoring painful

This is not because it's "bad" — it's because **execution is cheaper than planning** for language models.

Your job is to:

> **explicitly separate "planning mode" from "execution mode."**

This mirrors the **planner/executor split** you just built in your micro-agent.

---

### ☑ Concept

You should treat AI collaboration as a *two-phase protocol*:

**Phase 1 — Planning**

- clarify intent
- surface assumptions
- define scope
- agree on interfaces
- enumerate steps

**Phase 2 — Execution**

- implement exactly what was planned
- do not introduce new scope
- respect constraints
- produce reviewable output

The mistake most people make is letting these phases blur.

## 📌 Example 1 — What Happens Without Separation (Bad)

```
Build a data cleaning pipeline in Python.
```

Likely outcome:

- planning + coding mixed together
- assumptions hidden in code
- unclear extension points
- hard to review or refactor

---

## 📌 Example 2 — Explicit Planning Phase (Good)

```
You are acting as a junior data engineer.

Goal:
Design a data cleaning pipeline for tabular CSV data.

Current task:
PLANNING ONLY — do not write any code.

Instructions:
1) Propose a high-level pipeline outline (step names + brief descriptions)
2) Identify assumptions about the data
3) Clearly mark which steps are in scope vs out of scope for the first iteration
4) Ask clarifying questions if anything is ambiguous

Stop after the plan and wait for approval.
```

Output = text only, no code

---

## 📌 Example 3 — Execution Phase (After Approval)

```
Using the previously approved plan:

Current task:
EXECUTION ONLY — do not change scope or architecture.

Instructions:
1) Implement the `load_data` and `validate_schema` steps only
2) Use pandas
3) Return modular, testable functions
```

```
4) Do not implement cleaning or modeling yet

Deliverable:
Python code only
```

**The One Debugging Rule to Keep**

> **If AI can't explain why a bug occurs, it shouldn't be allowed to fix it.**

This single rule will prevent **~80%** of AI-assisted debugging failures.

---

**Why This Matters for AI-Driven Data Science**

This separation gives you:

- safer refactoring
- clearer diffs between iterations
- easier stakeholder communication
- alignment with agile sprint reviews
- better long-term code quality

It also makes AI:

- more predictable
- easier to correct
- less likely to hallucinate features

---

## Strong Analogy (Ties Back to Module 1)

| Module 1 Concept | Prompting Equivalent |
|---|---|
| BFS planning | Planning prompts |
| DFS execution | Execution prompts |
| Cache/memoization | Reusing approved plans |
| Agent orchestration | You (the human) |

You are acting as the **meta-agent**.

---

## Mini-Exercise 2.2

You'll write **two prompts** for the *same task*.

**Task context** You are building a data science pipeline for a tabular dataset.

---

**Part A — Planning Prompt**

Write a prompt that:

- explicitly forbids code
- asks for a pipeline outline
- surfaces assumptions
- identifies in-scope vs out-of-scope steps

---

**Part B — Execution Prompt**

Write a follow-up prompt that:

- references the approved plan
- limits scope to two steps
- forbids architectural changes
- specifies the exact deliverable

---

**Submission Format**

Reply with:

```
**Planning Prompt**
...

**Execution Prompt**
...
```

---

## ◇ MODULE 2.3 — Scoping AI Work for Agile Sprints

> **Theme: Turning prompt-level control into sprint-sized, reviewable AI work**

---

### Where Module 2.3 Sits (Re-grounding)

- **Module 2 (this module):** How you collaborate with AI inside a sprint
- **Module 3 (later):** How a team delivers across multiple sprints with AI

So Module 2.3 answers:

> "Given a sprint-sized goal, how do I scope AI work so it produces something reviewable, demoable, and safe?"

---

### 1 Concept: Sprint-Safe AI Scoping

A sprint-safe AI task has four properties:

1. **Small surface area**

2. **Explicit boundaries**
3. **Reviewable artifacts**
4. **No hidden architectural changes**

If any one of these is missing, AI output becomes risky.

---

## 2 The Sprint Slice Pattern (Core Technique)

You already know this intuitively — now we formalize it.

**The pattern**

```
Full Vision (context only)
↓
Sprint Slice (what we're actually doing)
↓
Acceptance Criteria (how we know it's done)
↓
AI Prompt(s) (planning → execution)
```

**Important:** The sprint slice is *not* a random subset — it's a **vertical slice** that produces something usable.

---

## 3 Concrete Example (Data Science Pipeline)

### ❌ Bad sprint scope (horizontal slice)

> "Implement all data cleaning logic."

Problems:

- ambiguous
- hard to demo
- unclear completeness
- easy to overbuild

---

### ☑ Good sprint scope (vertical slice)

> "Load a CSV, validate schema, and produce a validated dataset artifact."

Even though it's early in the pipeline, it's:

- end-to-end
- testable
- demoable
- extendable

---

## 4 Acceptance Criteria (This Is the Secret Weapon)

> AI should **never** be asked to "just build something" in a sprint.

Instead, you define **acceptance criteria** *before* execution.

**Example acceptance criteria**

```
Done means:
- CSV can be loaded from disk
- Missing required columns raise a clear error
- Output is a pandas DataFrame
- Functions are pure and deterministic
- No modeling or feature engineering exists
```

This does two things:

1. Prevents scope creep
2. Makes review objective

---

5 How This Looks as Prompts (Putting It Together)

Planning Prompt (Sprint-aware)

```
You are acting as a junior data engineer.

Sprint goal:
Produce a minimal, validated dataset ingestion step.

Context (future work, not now):
This pipeline will later include cleaning, feature engineering, and modeling.

Current sprint scope:
- Load CSV data
- Validate required columns

Out of scope:
- Cleaning
- Feature engineering
- Modeling

Acceptance criteria:
- Raises clear errors for invalid schema
- Returns a pandas DataFrame
- No architectural decisions beyond these steps

Task:
Propose a plan for this sprint slice only.
```

```
No code. Wait for approval.
```

Notes

- Looks like there's a couple ways to go about developing prompts within a sprint environment
  - Big picture first ( Goals: ), sprints second ( Current sprint scope: )
    - Hey I need to solve this problem, let's start here
  - Sprints ( Current sprint scope: ) first, with clear picture of what solution will be backing out to ( Context (future work, not now) )
    - This is what we're doing today, keep in mind these are the things we'll be doing later on

**Why This Maps So Cleanly to Agile Thinking (Without Needing Scrum Jargon)**

Even without naming Scrum explicitly, you're thinking in exactly the right terms:

**Older framing (goal-first)**

- "Here's the destination"
- Easy to accidentally overbuild
- Encourages waterfall thinking

**New framing (now-first, later-aware)**

- "Here's what matters this sprint"
- "These other things exist — do not touch them yet"
- Creates clean stopping points
- Encourages safe iteration

That's why:

- `Goal:` becomes background context
- `Sprint goal:` becomes the primary anchor
- Ordering matters: **now** → **later**, not **later** → **now**

---

**Ordering Matters More Than People Realize**

You also noticed the **ordering** of sections changed. That's critical.

**Old ordering (conceptual):**

1. Goal
2. Task
3. Instructions

**New ordering (temporal):**

1. What we're doing *now*
2. What exists *later*
3. What is explicitly *out of bounds*

4. How we know we're done

This helps AI (and humans) avoid:

- scope bleed
- architectural drift
- premature optimization

It's the prompting equivalent of a clean sprint backlog.

---

**Execution Prompt (Sprint-safe)**

```
Using the approved sprint plan:

Implement only:
- load_data
- validate_schema

Constraints:
- Python 3.10
- pandas only
- No new dependencies
- No refactors outside this slice

Deliverable:
Python code only.
```

This is **agile-compatible AI usage**.

---

## 6 Why This Is Not Module 3 Yet

In Module 2.3:

- You are still writing prompts
- You are still thinking sprint-by-sprint
- You are still controlling *individual AI tasks*

You are **not yet**:

- managing backlogs
- sequencing sprints
- communicating delivery risk
- coordinating humans + AI

That's Module 3.

Takeaways

- Prompt development transitions from static (*Goal/Current Task*) to time-conscious (*Sprint Goal/Context*) approach with roughly the same scoping parameters -> project management perspective
    - Current task first, big picture second -> introduces more opportunities to pivot as needed as opposed to defining goal first (much harder to refactor planning because AI will have already taken certain optimizations into account)
- Implement functionality first as simple, explicit functions and defer architectural abstraction until there is real pressure to do so
    - *"Functional core, architectural shell later."*
        - AI helps build the functional core quickly
        - Humans decide when and how to harden it architecturally
- **Practical Rule of Thumb**
    - If you can explain the task in one sentence, write functions.
    - If you keep repeating the same logic in three places, consider refactoring.
    - If state or lifecycle matters, consider classes.
- *"Delay architecture until behavior is stable."*

---

## Mini-Exercise 2.3

**Task Context**

You are building a data science pipeline that will eventually include:

- ingestion
- cleaning
- feature engineering
- modeling
- evaluation

---

**Your Task**

Define one **sprint-sized AI task** by writing:

1. **Sprint Goal** (1–2 sentences)
2. **In Scope** (bullet list)
3. **Out of Scope** (bullet list)
4. **Acceptance Criteria** (3–5 bullets)

You do **not** need to write prompts yet — just the sprint definition.

---

**Submission Format**

```
**Sprint Goal**
...

**In Scope**
- ...
```

```
**Out of Scope**
- ...

**Acceptance Criteria**
- ...
```

---

# ◇ MODULE 2.4 — Refactoring with AI (Safe vs Dangerous)

> **Theme:** Knowing **when** to refactor, **what** to refactor, and **how** to ask AI to do it safely.

---

## Why This Module Matters So Much

Most real-world AI failures don't happen when AI writes *new* code.

They happen when:

- AI refactors working code
- interfaces change silently
- assumptions are broken
- tests are skipped
- stakeholders lose confidence

So Module 2.4 is about **controlling change**, not just producing code.

---

## 1️⃣ Concept: Not All Refactors Are Equal

There are two fundamentally different kinds of refactoring:

### 🟢 Safe Refactors (Behavior-Preserving)

- renaming variables
- extracting helper functions
- removing duplication
- adding type hints or docstrings
- reorganizing files *without changing logic*

These are ideal for AI.

---

### 🔴 Dangerous Refactors (Behavior-Changing)

- changing function signatures
- introducing classes or frameworks
- modifying data formats
- altering execution order
- "improving" performance without benchmarks

These require **human judgment first**.

---

## 2 The Cardinal Rule of AI-Assisted Refactoring

> **AI should never refactor code unless it can explain *exactly* what will and will not change.**

This mirrors how you'd review a PR from a junior engineer.

---

## 3 The Safe Refactor Protocol (Use This Every Time)

When you ask AI to refactor, your prompt should enforce this sequence:

1. **Explain intent**
2. **List proposed changes**
3. **Declare what will not change**
4. **Wait for approval**
5. **Then apply the refactor**

If any step is skipped, the refactor is unsafe.

---

## 4 Concrete Example (Based on Your Pipeline)

### ❌ Dangerous prompt

```
Refactor this pipeline to be cleaner and more modular.
```

This invites:

- architectural changes
- hidden behavior changes
- scope creep

---

### ☑ Safe refactor prompt (planning mode)

```
You are acting as a junior engineer.

Context:
This code is currently working and covered by tests.

Task:
Propose a refactor to improve readability and reduce duplication.

Constraints:
- Do not change function signatures
- Do not change behavior
```

```
- Do not introduce classes or frameworks
- Do not add new dependencies

Instructions:
1) Describe the refactor in plain English
2) List exactly what files and functions would change
3) Explicitly state what will remain unchanged

Do not write code yet.
```

**1) Sprint-compatible safe refactor prompt (planning mode)**

This follows the Module 2.3 framing (now vs later, in/out of scope, acceptance criteria) while still being "no-code planning."

```
You are acting as a junior engineer.

Sprint goal:
Refactor the existing ingestion + schema validation code to improve readability
and maintainability,
without changing behavior.

Context (future work, not now):
This pipeline will later expand to cleaning, feature engineering, and model
training.

In scope (this sprint):
- Reduce duplication
- Improve naming and docstrings
- Extract helper functions where it reduces complexity
- Add type hints where they clarify interfaces
- Keep the current module/file structure unless a move is purely organizational

Out of scope:
- Any changes to runtime behavior
- Any changes to public function signatures
- Introducing classes/frameworks (functions only)
- Adding new third-party dependencies
- Performance optimizations unless measured and approved

Acceptance criteria:
- All existing tests pass unchanged
- Public function signatures remain identical
- Outputs for the same inputs are unchanged (behavior-preserving)
- Refactor plan lists exact files/functions to change
- Refactor plan explains why each change improves maintainability

Task (PLANNING ONLY):
1) Propose a refactor plan (no code)
2) List exact changes by file/function
```

```
3) State explicitly what will NOT change
4) Identify the highest-risk change (if any) and how to mitigate it
Stop after the plan and wait for approval.
```

**Key point**

Refactoring fits into a sprint when you frame it like:

- **goal** (maintainability)
- **constraints** (behavior unchanged)
- **acceptance criteria** (tests + interface stability)

That's sprint-safe.

Notes

- Notice under *Out of scope* the bullet *Introducing classes/frameworks (functions only)*, coming back to earlier questions this will be a repeating bullet to have while working to develop entirety of initial pipeline.

---

**2) Example prompt for "architecture/productization sprint" after functionality is done**

You're right: once the pipeline works, teams often need to *host* it:

- as a CLI tool
- as a scheduled job
- as a service (API)
- as part of an application

Here's how the collaboration changes: you move from "implement logic" to "design interfaces and deployment shape."

**Architecture sprint (planning prompt)**

```
You are acting as a junior software engineer helping productize an existing data
pipeline.

Sprint goal:
Propose a minimal architecture to run the completed pipeline reliably in
production.

Context:
- The pipeline functionality already exists and is tested.
- Stakeholders need a runnable entry point and a deployment shape.
- We want a minimal design that can evolve.

In scope (this sprint):
- Choose ONE hosting form for MVP (CLI OR scheduled job OR simple API) and justify
it
```

```
- Define module boundaries (where pipeline logic lives vs orchestration)
- Define configuration approach (e.g., config file/env vars)
- Define logging and run reporting outputs (human + machine readable)
- Define how the pipeline is invoked end-to-end

Out of scope:
- Rewriting pipeline logic
- Building full UI
- Adding complex orchestration frameworks
- Full cloud deployment IaC
- Auth, multi-tenancy, or scaling concerns beyond MVP

Constraints:
- Keep pipeline code unchanged unless strictly necessary
- Prefer standard library + existing deps
- Favor clarity over over-engineering

Acceptance criteria:
- Architecture proposal includes a diagram (ASCII is fine)
- Clear list of components and responsibilities
- Clear “entry point” definition and example usage
- Explicit risks/tradeoffs and why this MVP is appropriate

Task (PLANNING ONLY):
Provide 2 architecture options (MVP + slightly more robust), recommend one, and
explain why.
No code. Wait for approval.
```

**What changed vs earlier prompts?**

- You’re no longer asking for *data logic*.
- You’re asking for:
  - **interfaces**
  - **entry points**
  - **operational concerns** (logging, config, deployment shape)
- And you’re explicitly protecting existing functionality (“do not rewrite pipeline logic”).

That’s the architectural version of “safe refactor.”

---

☑ **Execution prompt (after approval)**

```
Using the approved refactor plan:

Apply the refactor exactly as described.

Constraints:
- Preserve all existing behavior
- Keep all public interfaces identical
- No new abstractions
```

```
Deliverable:
Code diff only.
```

This is **how professionals use AI safely**.

---

## 5 How This Connects to Your Earlier Question

You asked earlier whether it's common to:

> *build functionality first, then refactor later*

Module 2.4 formalizes that instinct.

**Healthy workflow:**

1. Sprint delivers working functionality
2. Tests pass
3. Stakeholders approve behavior
4. **Only then** do you consider refactoring
5. Refactor is scoped, reversible, and reviewed

AI fits beautifully into *steps 4–5* when guided correctly.

---

## 6 Refactoring Triggers (When to Consider It)

Good reasons to refactor:

- duplicated logic appears
- function bodies exceed cognitive limits
- interfaces stabilize
- tests exist

Bad reasons:

- "AI suggested it"
- "It feels messy"
- "We might need this later"

Takeaways

- Refactors that are *Behavior-Preserving* are ideal for AI whereas anything *Behavior Changing* requires human judgement first (e.g., don't just rely on AI recommendations)
    - **Behavior Preserving:** cleaning function documentation, renaming variables, reorganizing files without changing logic
        - Example: in Module 1's capstone planning & task execution logic, while left in tact, was moved from `agent.py` to their own respective `.py` files

    - **Behavior Changing:** changing function signatures, introducing classes or frameworks, modifying data formats, altering order execution, "improving" performance without benchmarks
- "AI should never refactor code unless it can explain exactly what will and will not change."
    - AI needs to assess impact of refactor (planning) before executing plan -> provides opportunity

---

## Mini-Exercise 2.4

**Task Context**

You have a working data ingestion + schema validation step (like the one you scoped in Module 2.3).

---

Your Task

Write **two prompts:**

---

**Prompt A — Refactor Planning Prompt**

Must:

- forbid code
- explain refactor intent
- list constraints
- require explanation of what won't change

---

**Prompt B — Refactor Execution Prompt**

Must:

- reference the approved refactor plan
- forbid behavior changes
- specify deliverable format

---

**Submission Format**

```
**Refactor Planning Prompt**
...

**Refactor Execution Prompt**
...
```

---

# ◇ MODULE 2.5 — Review, Validation, and Trust-Building

> **Theme:** Turning AI output into **reviewable, defensible, stakeholder-ready work**

**Module 2.5** is the capstone mindset for *trust*. Everything you've done so far (scoping, planning, refactoring) only matters if **other humans can trust the results**.

This module answers:

> *"How do I review, validate, and communicate AI-assisted work so it's safe to ship?"*

---

## Why This Module Exists

AI doesn't remove responsibility — it **moves responsibility upstream**.

Stakeholders don't ask:

- "Did AI generate this?"

They ask:

- "Can we trust this?"
- "What changed?"
- "What could break?"
- "How do we know it works?"

Module 2.5 gives you a **repeatable review protocol** so AI-assisted work:

- passes code review
- survives refactors
- earns stakeholder confidence

---

## 1 Concept: AI Output Is a Draft, Not a Decision

The core rule:

> **AI produces candidates. Humans approve decisions.**

So your job is to ensure every AI-assisted change:

- can be reviewed
- can be validated
- can be explained

---

## 2 The Three Pillars of Trust

Every AI-assisted sprint deliverable should satisfy **all three**:

### 🧱 Pillar 1 — Reviewability

- Small diffs
- Clear intent
- Explicit scope
- No hidden changes

### 🩹 Pillar 2 — Validation

- Tests (existing or new)
- Deterministic behavior
- Known failure modes
- Clear error messages

### 📣 Pillar 3 — Explainability

- What changed?
- Why it changed?
- What didn't change?
- What risks remain?

If any pillar is missing, trust erodes.

---

## 3 The AI Review Checklist (Use This Every Time)

When reviewing AI-assisted work, ask:

1. **Scope**
   - Does this match the approved plan/sprint?
   - Did AI introduce anything extra?
2. **Behavior**
   - Are inputs/outputs unchanged?
   - Do tests still pass unchanged?
3. **Interfaces**
   - Are public function signatures identical?
   - Are callers unaffected?
4. **Risk**
   - What could break?
   - Is the risk acceptable for this sprint?
5. **Artifacts**
   - Can I explain this change to a stakeholder in **2 minutes**?

This checklist is your safety net.

---

## 4 Concrete Example: Review Prompt for AI Output

After AI generates code, you should *not* just eyeball it. Instead, you can ask AI to help you review — but in a **bounded way**.

### ☑ Safe review prompt

```
You are acting as a code reviewer.

Context:
```

```
This code was refactored to improve readability without changing behavior.

Task:
Review the following diff.

Checklist:
- Confirm no behavior changes
- Confirm function signatures are unchanged
- Identify any risky or unclear changes
- Suggest test cases if something seems under-tested

Do NOT rewrite code.
Do NOT propose additional refactors.
Return findings as bullet points only.
```

This keeps AI in *review mode*, not *editor mode*.

---

## 5 Stakeholder-Facing Summary (This Is Huge)

A **3–5 sentence** summary can make or break trust.

**Example "AI-assisted sprint summary"**

```
This sprint focused on improving the maintainability of the data ingestion step.

No functional behavior changed.
All existing tests passed unchanged.

Changes were limited to:
- clearer function naming
- reduced duplication
- added type hints

Risk is low. No downstream consumers were impacted.
```

You'll notice:

- no AI jargon
- no hype
- just outcomes and risk That's intentional.

Q: Even with excellent prompting, does AI hallucinate during review process? What does this look like?

Appendix

- Under Module 2, see *What to Watch Out For with AI Sprint Review Summaries*

A: **Even well-prompted AI can hallucinate — but at review time, hallucinations are usually detectable, not subtle.**

A: Hallucinations during review tend to follow recognizable patterns

- Confidence without Evidence
  - AI confidently states something is correct without pointing to specific lines or behaviors
  - **Watch for:** claims without citations, general statements instead of line-level reasoning, no mention of uncertainty or risk
  - **Counter:** Require evidence
    - "*For each claim, reference the specific function or diff section that supports it.*"
- Phantom Change
  - AI claims a change happened that did not actually occur
  - **Watch for:** review comments that sound plausible but don't match the diff, mentions of changes you don't remember approving
  - **Counter:** force grounding
    - "*Only comment on changes that appear in the provided diff. If no change exists, say 'no change observed.*'"
- Speculative Risk
  - AI invents risks that are technically possible but not relevant to the change
    - "This could cause race conditions in concurrent execution."
    - except code is synchronous, so no concurrency exists...
  - **Watch for:** risks that don't connect to the current code path, mentions of systems (threads, async, memory) not present in the code
  - **Counter:** scope the risk lens
    - "*Only identify risks introduced by this change, not hypothetical future extensions.*"
- Over-Helpful Suggestion Drift
  - AI starts proposing improvements instead of reviewing
    - "You could improve this by introducing a base class or pipeline framework."
  - **Watch for:** "*you might consider...*", "*it could be improved by...*", architectural suggestions during review
  - **Counter:** hard role lock
    - "*Do not suggest improvements. Only identify issues or confirm correctness.*"

---

**A Meta-Rule That Helps a Lot**

> **If the AI review sounds like a blog post, it's hallucinating. If it sounds like a cautious PR comment, it's probably grounded.**

Professional reviews are:

- boring
- specific
- constrained
- slightly cautious That's what you want.

**One Extra Defensive Trick (Very Effective)**

After an AI review, ask **one follow-up question**:

```
What is the single most uncertain claim in your review?
```

A grounded review will respond thoughtfully. A hallucinated review often collapses or restates itself.

Appendix

- Under Module 2, see *How to Approach AI-Supported Debugging*

Prompts for Debugging Principles

- Debugging is not refactoring
  - Goal of debugging is to *identify cause, verify hypothesis, apply minimal fix*
- Force hypothesis before fix
  - AI is very good at proposing fixes but less reliable at identifying root causes unless forced
- Minimize blast radius
  - Always constrain *files allowed to change, functions allowed to change, interfaces that must remain identical*
- Debug in two phases -> just like planning/execution
  - Phase A: Diagnosis (no code)
  - Phase B: Fix (minimal diff)

Takeaways

- Prompts written are for: *planning, execution, review, debugging,* and *executive summary*
- AI can hallucinate review summaries, be weary while reviewing status (4 forms of hallucination)

---

## 6 How This Completes Module 2

You now have a full **AI collaboration loop**:

1. **Scope the work** (2.3)
2. **Plan explicitly** (2.1, 2.2)
3. **Execute safely** (2.2)
4. **Refactor responsibly** (2.4)
5. **Review + communicate clearly** (2.5)

This loop is what makes AI usable in real data science projects.

---

## Mini-Exercise 2.5

**Task Context**

You just completed a **safe refactor** of a data ingestion + schema validation step using AI.

---

**Your Task**

Write **two short artifacts:**

---

**A) Review Checklist Summary**

3–5 bullet points confirming:

- scope adherence
- behavior preservation
- interface stability
- remaining risk (if any)

---

**B) Stakeholder Update**

3–5 sentences that:

- explain what was done
- explain what was not done
- communicate risk clearly
- avoid AI jargon

---

**Submission Format**

```
**Review Checklist Summary**
- ...

**Stakeholder Update**
...
```

---

## What Comes After This

Once we complete this exercise:

- **Module 2 is complete**
- We'll do a **Module 2 Capstone** (short, practical)
- Then we'll move into **Module 3 — Managing AI-Driven Projects at Scale**