AI-Assisted Engineering Workflow

Observe

/observe build context
YOU build mental modal

Bug

Debug

/debug suggests hypothesis, proves with tests

Regular debugging

YOU mirror with your own debugging

Feature

Is system structure clear for this feature?

Plan

YOU break down feature, iterate plan

Implement

/implement writes code/tests, **YOU** review every change

Test

/test writes comprehensive tests; **YOU** run, monitor, correct

Review

YOU do Line-by-line review; **/code-review** for secondary scan

Refactor

Consult

Discuss, critique refactor approaches

Decide

Figure out the best approach; **Never let AI** refactor real code alone

Professional AI-Assisted Software Development

A practical, battle-tested approach to using AI as a coding partner—not a replacement—in professional software engineering.

Core Philosophy

AI is a mid-level engineer, not a senior architect. It excels at mechanical execution, pattern recognition, and repetitive problem-solving, but struggles with independent judgment, system-level decisions, and context navigation. The developers who see 3-5x productivity gains don't treat AI as autonomous; they treat it as a highly capable collaborator that requires skilled direction.

This guide establishes three critical constraints for professional AI usage:

1. **You maintain cognitive ownership**: You must understand the code at line-by-line depth
2. **AI works inside your context**: You set the boundaries, pace, and direction
3. **Multitasking kills productivity**: Context switching between AI sessions destroys the cognitive load benefits that make AI valuable

Research shows that developers who maintain focus on a single AI collaboration session show 40% better code quality and 2x faster iteration cycles compared to those juggling multiple AI tasks simultaneously.

Project Documentation: Keep It Tool-Agnostic

Separate **project knowledge** (what the system does and why) from **AI instructions** (how AI should work).

Documentation Structure

```
docs/
|-- system_patterns.md
|-- tech_context.md
|-- testing.md
|-- formatting.md
|-- conventions.md
|-- backend-workflow.md
`-- add-[component]-workflow.md
```

Critical rule: These docs are shared with your team through version control. They're not AI-specific instructions—they're project memory that happens to also guide AI.

When updating docs as part of merged code:

- Add new patterns after implementing them (AI learns from your code)
- Document architectural decisions when making them
- Update examples when patterns evolve
- Never maintain docs separately from code

The Workflows

All workflows follow a strict principle: **you lead, AI executes**. Every AI action is initiated by you, monitored by you, and corrected by you immediately.

Main Workflow: Feature Implementation

OBSERVE PHASE **Your job**: Build mental model of existing codebase

- Walk through relevant services and how they connect
- Understand existing patterns and constraints
- Identify where new code fits

AI's job: Match your understanding

- Use `/observe` command (or invoke directly):

  Option 1 - Slash command:

  ```
  /observe auth service and user service
  ```

  Option 2 - Direct prompt:

  ```
  Your task is to **observe and understand** the codebase for the subject

  ## Guidelines

  1. **Exploration Only** - Do NOT:
     - Debug specific issues
     - Create plans for new features
     - Propose fixes or changes
     - Generate implementation tasks

  2. **Focus On** - DO:
     - Explore existing patterns and design decisions
     - Map relationships between components
     - Understand architectural choices
     - Document data flows and dependencies
     - Identify conventions and practices
     - Note how different modules interact

  3. **Update Docs**:
      - Read docs/ folowing @docs.mdc rule
      - Check if any of the docs needs updating or extending and only put new information to any of t

  Focus on: auth service in @docs/system_patterns.md and src/services/auth.ts, and user service
  ```

- Ask AI questions: "How do these services currently handle X?" "Where would the new code fit?"
- Consult on your high-level ideas:

  ```
  I'm thinking of implementing [your approach].

  Use @docs/system_patterns.md.
  Do you see any issues with this approach given our existing patterns?
  What questions do you have about my thinking?
  ```

When to pause here: If the system doesn't have clear structure for this feature to fit naturally, **don't proceed**. Spend time now establishing patterns and structure. AI can't fix architectural gaps by coding—it can only amplify them.

This phase typically takes the most time and effort, but saves exponentially more downstream.

PLAN PHASE **Your job**: Design the implementation at all detail levels

- If the feature is too large for you to hold in your head completely, **break it down**. Implement one component at a time.
- Iterate on the plan with AI:

```
Based on your understanding from the OBSERVE phase:

I want to implement [component].

Create a detailed plan that includes:
1. Files affected
2. New database schema or changes
3. New API endpoints
4. How it connects to existing services
5. Error cases and edge cases

Use @docs/system_patterns.md for architectural decisions.
```

- **You decide if the plan is right**: "Does this match how I want to build this?" If yes, continue. If not, refine.
- Iterate until you know exactly how every detail will be implemented and you're on board with it.

Research shows: Teams spend 40-60% of total effort in planning but save 70% of rework downstream. This is where productivity gains happen.

IMPLEMENT PHASE Use Test-Driven Development (TDD)

AI performs exceptionally well at TDD because tests provide explicit success criteria. Teams that adopted TDD specifically for AI work (even if never done before) report 100% success rates on refactoring phases and zero regression rates.

1. **AI writes tests** (AI excels here):

   Option 1 - Slash command:

   ```
   /test [component]
   ```

   Option 2 - Direct prompt:

   ```
   Write comprehensive tests for the specified code/feature/bug fix.

   ## Workflow

   1. **MUST ALWAYS Read @docs/testing.md** to understand how to write/execute tests
   2. First check for existing tests for a subject and if possible - extend them to cover new use case
   3. **Run tests** to verify they pass

   Focus on: [component]

   Test cases:
   - Happy path: [describe success]
   - Error cases: [describe failures]
   - Edge cases: [list boundaries]

   DO NOT implement the feature yet. Only write tests.
   ```

   - Run tests with you present: `npm run test:unit`
   - Confirm they fail (they should—no implementation yet)
   - Commit: "test: add [component] tests"

2. **AI implements** (AI executes; you monitor):

   Option 1 - Slash command:

   ```
   /implement [component feature description]
   ```

   Option 2 - Direct prompt:

   ```
   1. **implement** the requested feature or code change, MUST FOLLOW documentation @docs.mdc.

   2. Test implementation with @test.md command

   3. **Before Finishing**:
      - Run linter and fix any errors
      - Run formatters if any documented in docs/formatting.md
      - Verify tests pass
      - Check that code follows project conventions
      - Ensure no debug code or comments left behind

   Implement: [component] to pass all tests
   Follow:
   - Patterns from @docs/system_patterns.md
   - Naming from @docs/conventions.md
   - Error handling patterns from @docs/system_patterns.md
   ```

3. **You review code immediately**: Read the output line-by-line. If something doesn't make sense:
   - **STOP AI** (don't let it continue iterating on something wrong)
   - Ask AI to explain the problematic section
   - Either accept the explanation or ask AI to rewrite it
   - Your cognitive understanding must stay 100% intact

Critical rule: The moment you can't maintain your mental understanding of what AI is outputting, **stop**. Switch AI off and continue yourself, or reset and get back on track.

Organizations that maintained line-by-line oversight achieved 40% better code quality compared to those who "trusted" AI output.

TEST PHASE

1. **If you used TDD**: Run full test suite:

   ```
   npm run test:unit
   npm run test:integration (if applicable)
   npm run type-check
   npm run lint
   ```

   All should pass.

2. **If you didn't use TDD**: Ask AI to write comprehensive tests:

   Option 1 - Slash command:

   ```
   /test [implementation we just did]
   ```

   Option 2 - Direct prompt:

   ```
   Write comprehensive tests for the specified code/feature/bug fix.

   ## Workflow

   1. **MUST ALWAYS Read @docs/testing.md** to understand how to write/execute tests
   ```

```
2. First check for existing tests for a subject and if possible - extend them to cover new use case
3. **Run tests** to verify they pass

Focus on: [implementation we just did]
- All happy paths
- All error conditions
- All edge cases (null, undefined, empty arrays, boundary values)
```

- Run tests: Watch for failures
- Have AI fix bugs until tests pass
- Your job: Stay line-by-line with the code, catching bad assumptions

Warning: AI gets stuck in loops when it can't find bugs. Stay engaged and help it debug by pointing to specific lines where assumptions break down.

REVIEW PHASE Clear AI context or start new session. Fresh eyes catch more.

- You review code line-by-line exactly as you would for a junior developer
- Look for: Logic errors, edge cases, performance issues, security issues, architectural alignment
- Then ask AI for second review:

  Option 1 - Slash command:

  ```
  /clear (or start new session)

  /code-review
  ```

  Option 2 - Direct prompt:

  ```
  Review code changes for critical issues only.
  Not only review the diff itself, but go deep into the context code to understand how it works under

  ## Get Changes

  Use git read commands:
  - `git status` - see modified files
  - `git diff` - unstaged changes
  - `git diff --staged` - staged changes
  - `git diff main...HEAD` - all branch changes vs main

  ## Check Patterns

  Reference @docs.mdc to understand existing patterns and conventions. Identify if changes break esta
  - Architectural patterns
  - Design decisions
  - Naming conventions, code formatting
  - Module relationships

  ## Review Criteria

  Focus on **critical issues only**:
  - **Bugs** - Logic errors, null handling, edge cases
  - **Performance** - N+1 queries, inefficient algorithms, memory leaks
  - **Security** - SQL injection, XSS, auth bypasses, data exposure
  - **Correctness** - Business logic errors, data integrity violations
  - **Pattern Breaks** - Violations of established architectural patterns or conventions
  ```

Bug Workflow

AI can be hit-or-miss for debugging. The trick: **have it prove every assumption with a test**.

OBSERVE PHASE Same as main workflow—build your mental model of the system and have AI match it.

DEBUG PHASE Option 1 - Slash command:

```
/debug [bug description or context]
```

Option 2 - Direct prompt:

```
**Your task**: Find and prove the root cause. **Do NOT fix it yet.**

## Process

1. **Debug** -  identify the most likely cause of a bug
2. **Prove** - Write a unit test (follow `@docs/testing.md`) that proves the issue
3. **Run** - Execute the test to confirm your hypothesis

if test doesn't prove you are correct with the cause - continue with your observation untill you find Al

4. **Report** - Respond with a short summary only

## Output Format

ROOT CAUSE: [Brief description]

TEST RESULT: [Pass/Fail status]

EVIDENCE: [1-2 sentences explaining how the test proves the cause]

Do not propose fixes. Only prove the diagnosis.

Bug: [description or stack trace]
Working context: [what were you trying to do]
```

Your job: Start regular debugging in parallel. This maintains your cognitive understanding and catches future problems early. AI can narrow the search space; you close the gap.

IMPLEMENT + TEST + REVIEW Same phases as main workflow.

Refactor Workflow

Critical rule: Never let AI refactor real code independently.

Refactoring requires understanding tradeoffs, system context, and long-term consequences. AI can help you think about refactoring, but shouldn't execute it unsupervised.

OBSERVE PHASE Understand the current code and constraints.

CONSULT PHASE Work inside AI's context like consulting with a coworker:

```
I'm considering refactoring [component].

Read @docs/system_patterns.md and [component code].

Here are my ideas:
1. [Your approach 1]
2. [Your approach 2]

Which approach is better and why?
What problems do you see with my thinking?
```

- AI critiques your ideas
- You decide which direction to take
- **This way, end decisions are yours, and you maintain cognitive load**

Then execute:

- Write tests for current behavior first
- Implement refactoring following IMPLEMENT phase
- Tests should pass before and after

Why this works: Pair programming against "agentic dementia." When AI makes autonomous decisions, you lose context. When you make decisions with AI's input, you stay in the driver's seat.

Multitasking Destroys Productivity

Context switching between multiple AI sessions—one writing tests, another implementing, a third reviewing—kills the cognitive load benefits that make AI valuable.

Research finding: Developers maintaining focus on a single AI session showed 40% better code quality and 2x faster iteration cycles.

Rule: One task, one AI session, full attention. When you finish, commit. Then start next task with fresh context (new session, AI reads docs).

Background agents don't fit professional SWE because:

- You lose cognitive model of what AI is doing
- Debugging becomes harder (context spread thin)
- Quality degradation accelerates with context diffusion
- You can't catch AI mistakes if you're not watching

What works instead:

- Focused 2-3 hour sessions per feature component
- Fresh AI session for next component
- AI reads docs to understand previous work
- You maintain full mental model throughout

Common Pitfalls

Pitfall 1: Skipping OBSERVE & PLAN

Jumping straight to implementation costs 5-7x more in rework. Professional teams spend 40-60% of effort planning. This saves 70% of rework.

Pitfall 2: Losing Cognitive Ownership

If you can't explain what AI outputted at line-by-line depth, you've lost control. Stop immediately. Read code, ask questions, or revert.

Evidence: Experienced developers (2.6% "highly trust" AI) are most cautious about this. They know it matters.

Pitfall 3: "Smart" Refactoring by AI

AI refactoring without human decisions introduces subtle bugs at scale. Enterprise teams report 4x better ROI when humans make refactoring decisions first, then AI executes.

Pitfall 4: Trusting Without Validation

46% of professional developers actively distrust AI accuracy. 97% say human verification is essential. Treat all output as potentially wrong until proven.

Pitfall 5: Large Context = Better Results

Counter to intuition, large context windows (2+ million tokens) sometimes produce worse results. One developer fed a project to Gemini Code Assist—it modified 31 files, introduced 260+ type errors, and broke the build. A smaller, focused Claude workflow completed the same refactoring in 11 minutes with zero errors.

Lesson: Use focused AI agents with relevant context, not one AI with everything.

Team Implementation

1. **Install setup**: One command, supports both Cursor and Claude Code
2. **Use shared docs**: Version-controlled `docs/` folder guides both team members and AI
3. **One workflow per project**: All team members follow OBSERVE → PLAN → IMPLEMENT → TEST → REVIEW
4. **Code review includes AI output**: Human review is critical, especially for complex logic
5. **Document patterns**: After implementing once, generate workflow doc for next time

---

Metrics That Matter

Professional teams track:

- **Cycle time**: Days from requirements to production (AI typically reduces by 15-25%)
- **Defect rate**: Bugs per 1000 lines (maintain same or improve)
- **Code review time**: Hours spent reviewing (human + AI review combined)
- **Rework percentage**: % of effort redoing code (should decrease with TDD + AI)

Don't track:

- Lines of code written (meaningless with AI)
- Suggestions accepted (wrong incentive)
- Velocity increases alone (can mask quality problems)

Bottom Line

AI is powerful when treated as a mid-level engineer that executes inside your context. It fails spectacularly when treated as autonomous or when you lose cognitive ownership.

The pattern that works:

1. You design at architecture level
2. AI executes at implementation level
3. You maintain full understanding throughout
4. One focused session per task
5. Human review, always

Teams following this pattern report:

- 3-5x faster complex feature development
- 40% better code quality
- Zero regression rates on refactoring
- Sustainable long-term productivity

Teams that skip steps (skipping planning, losing cognitive ownership, multitasking) report:

- Initial speed gains followed by quality collapse
- Increased technical debt
- Higher rework rates
- Burnout from context switching

Choose the former. Professional SWE hasn't changed—you still need to think hard. AI just makes execution faster when you're thinking clearly.