The AI Coding Loop: Verification-Driven Development with AI Agents

Modern software engineering with AI is less about "perfect prompting" and more about **disciplined process**. While AI can generate vast amounts of code instantly, the primary challenge shifts from authorship to **verification**. This guide outlines a repeatable workflow to ensure AI-generated code is secure, accurate, and maintainable — supported by real-world examples from active development.

The Core Problem: The "One-Shot" Trap

The "5-second high" occurs when an AI generates a complete module from a single sentence. However, this creates a **technical debt of understanding**. If you cannot verify the output, you do not own the code, making it a liability rather than an asset.

> **Rule of Thumb:** Treat AI output like code from a stranger — useful, but untrusted until proven by tests.

The Mindset Shift: Engineering over Prompting

In the AI era, your value shifts from typing speed to three core competencies:

1. **Problem Definition:** Defining the goal clearly.
2. **Decomposition:** Breaking large systems into small, testable "bricks."
3. **Verification:** Proving the result is correct via runnable constraints.

You are the Architect, AI is the Typist: You are entirely responsible for the logic, security, and data flow. The AI executes; you own the outcome.

The 7-Step Verification Loop

Never assume the agent's first output is flawless. Use this loop to guide AI incrementally rather than asking for a complete application at once:

1. **Define the Goal:** State the objective in one clear sentence.
2. **Establish Rules:** List the non-negotiable technical constraints (what must be true).
3. **Provide Examples:** Define expected Input ? Output mappings.
4. **Identify Edge Cases:** List "weird" or bad situations the system needs to handle.
5. **Request a "Small Piece":** Ask for a specific function or logic gate, not the whole system.
6. **Demand Tests:** Require the AI to provide runnable assertions.
7. **Iterate:** Treat failing tests as a "flashlight" to refine your next prompt and fix ambiguities in your rules.

Precise vs. Imprecise: The Power of Constraints

Your prompts must be **highly precise when it comes to rules, constraints, and edge cases**. Ambiguity is the enemy of secure AI-generated code.

Example: Server-Side Cart Calculator

If you simply ask an AI to "build a shopping cart," you risk getting vulnerable client-side logic where a user could manipulate prices. Instead, define a strict trust boundary where the server is the single source of truth:

- **Ignore Client Prices:** Never accept a price sent from the browser; use a server-side catalog.
- **Validation:** Quantity ? 1; Tax/Discount ? 0.
- **Order of Operations:** Apply discounts *before* calculating tax.
- **Precision:** Round money to 2 decimal places (or use cents/integers).

The "Golden Rule" Prompt Template

```
Goal: Calculate shopping cart totals.
Rules:
- Input: productId, qty.
- Source of Truth: Use internal PRODUCTS catalog.
- Constraints: qty >= 1; non-negative tax/discount.
```

```
- Math: Discount first, then tax; round to 2 decimals.
Examples: 2 T-shirts ($20) + 1 Mug ($12.50) = $52.50 subtotal.
Edge Cases: Unknown productId, qty = 0.
Deliver: One JS file with Node.js 'assert' tests.
```

Short vs. Long Prompts: The Iterative Workflow

Instead of writing one massive prompt, the most effective strategy is **iterative prompting**. Start with a structured, medium-length prompt to define the goal and constraints, then transition to short, highly focused commands to build and refine the output incrementally.

Example: Rapid UI Iteration via Short Prompts

During the development of a real-time events dashboard, a developer used extremely short prompts to polish the UI once the foundational context was established by the AI:

- The developer prompted: *"make the bar chart smaller and horizontal. different colors for channels. randomly assigned."*
- Because the AI already understood the established architecture, this short prompt was enough for the agent to formulate a detailed implementation plan — creating a compact horizontal layout and using a deterministic hash-to-color function so that channels kept a stable pseudo-random color across page reloads.
- The developer followed up with rapid micro-prompts like *"increase font contrast of labels of bar chart"* and *"the colors of the background and the overlays do not match the dark"*. The AI executed these perfectly by tuning CSS overlay tokens and applying theme-aware colors.

Actively Verifying System Logic

Verification isn't just about automated tests; it's also about actively questioning the AI's architectural decisions. When the AI added data filters to a dashboard's charts, the developer didn't just accept the code. They verified the logic by asking: *"how do the filters work? do they filter visible data or data on the server?"*

Only after the AI confirmed that the filters were applied securely on the server side via SQL query parameters did the developer instruct the AI to solidify this architecture: *"document the changes do*

you?", ensuring the verified logic was permanently recorded in the project's README.

Technical Implementation (Node.js)

The following implementation demonstrates the difference between "vulnerable" code and "engineered" code — applying the 7-Step Loop to a server-side cart calculator.

```
// cart.js - Run with: node cart.js
const assert = require("node:assert/strict");

const PRODUCTS = {
    tshirt: { name: "T-shirt", priceCents: 2000 },
    mug: { name: "Mug", priceCents: 1250 }
};

/**
 * CORRECT: Uses trusted catalog & validates inputs
 */
function cartTotal(cartItems, discountPercent = 0, taxPercent = 0) {
    if (!Array.isArray(cartItems)) throw new Error("Invalid input");

    let subtotalCents = 0;

    for (const item of cartItems) {
        const product = PRODUCTS[item.productId];
        if (!product) throw new Error("Unknown product: " +
item.productId);
        if (item.qty < 1) throw new Error("Invalid quantity");

        // Logic: Use PRODUCT.priceCents, NOT item.price
        subtotalCents += product.priceCents * item.qty;
    }

    const discountCents = Math.round(subtotalCents * (discountPercent /
100));
    const afterDiscount = subtotalCents - discountCents;
    const taxCents = Math.round(afterDiscount * (taxPercent / 100));

    return {
        subtotalCents,
        discountCents,
        taxCents,
        totalCents: afterDiscount + taxCents
    };
```

```
}

// Validation Test
const cart = [{ productId: "tshirt", qty: 2 }];
const result = cartTotal(cart, 10, 8);
assert.equal(result.subtotalCents, 4000);
console.log("Tests Passed: Verification Successful.");
```

Summary

- **AI is a Tool, Not an Architect:** You are responsible for the logic; the AI is the typist.
- **Failing Tests are Data:** If a test fails, it reveals an ambiguity in your rules.
- **Fundamentals Matter More:** Security, data flow, and edge-case thinking are now the primary skills of the developer.

TL;DR

This guide introduces **Verification-Driven Development**, a structured methodology for integrating AI into software engineering. The text argues that modern coding requires a shift from **manual authorship to rigorous oversight**, prioritizing problem decomposition and testing over simple prompting. To avoid untrusted code, the workflow uses a **seven-step iterative loop** designed to build software through small, verifiable increments. Central to this approach is treating AI as a subordinate tool while the human developer maintains responsibility for logic and security. By emphasizing **strict technical constraints** and edge-case identification, the generated modules become both accurate and maintainable. The developer's value now lies in **high-level system design** and the ability to prove that code functions correctly under pressure.

Revision #2

Created 2026-03-13 09:13:22 UTC by Carsten

Updated 2026-07-05 14:25:18 UTC by Carsten