The AI Coding Loop: A Guide to Verification-Driven Development

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.

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.

The 7-Step AI Coding Loop

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 technical constraints (what must be true).
3. **Provide Examples:** Define expected Input ? Output mappings.
4. **Identify Edge Cases:** List "weird" or "bad" situations to handle.
5. **Request a "Small Piece":** Ask for a specific function or logic gate, not the whole app.
6. **Demand Tests:** Require the AI to provide runnable assertions.
7. **Iterate:** Use failing tests as a "flashlight" to refine the prompt.

Practical Application: Server-Side Cart Calculator

A common beginner mistake is trusting client-side data. To build a secure shopping cart, we apply the loop to a specific sub-problem: **The Total Calculator.**

The Logic Pipeline

The calculator must follow a strict trust boundary where the server is the source of truth.

Key Constraints:

- **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 for precision).

The "Golden Rule" Prompt Template

Plaintext

```
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.
```

Technical Implementation (Node.js)

The following implementation demonstrates the difference between "vulnerable" code and "engineered" code.

JavaScript

```
// 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 for Wiki

- **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 artificial intelligence 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 the risks of untrusted code, the author outlines a **seven-step iterative loop** designed to build software through small, verifiable increments. Central to this approach is the **"Golden Rule" of verification**, which treats 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 workflow ensures that generated modules are both accurate and maintainable. Ultimately, the source highlights that a developer's value now lies in **high-level system design** and the ability to prove that code functions correctly under pressure.

Revision #1

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

Updated 2026-03-13 09:18:46 UTC by Carsten