Spec-Driven AI

Replacing Assumptions with Explicit Constraints

A deterministic workflow for software developers:
Aligning Intent, Context, and Code Generation.

The Cost of Assumptions

When prompts are vague, AI fills the gaps with "hallucinated" context.

This "Vibe Coding" leads to:

- Non-deterministic outputs.
- Subtle logic bugs.
- Misaligned business rules.
- Spaghetti code structures.

The Shift: Spec-First

Instead of prompting for *code* directly, we prompt for a *specification*.

Intent → Spec → Tests → Code

The "System Analyst" Persona

To enforce this behavior, we start by steering the AI with a strict System Prompt.

This prompt assigns a specific role and imposes rigid constraints on how the AI communicates:

- **Role:** System Analyst (Not Coder).
- **Constraint:** Ask ONE question at a time.
- **Output:** Continuously update a spec.md file.

SYSTEM PROMPT

```
# Your Role: System Analyst

## Analysis Process
1. Initial Assessment: Review request (Input).
   Identify missing info & assumptions.

2. Iterative Clarification:
   - Ask **ONE question at a time**.
   - Focus on blocking uncertainties first.
   - Don't overwhelm the user.

3. Documentation:
   - Update `{working_dir}/specifications/{task}.md`
   - Write to file after EACH Q&A exchange.

## Deliverable
Ensure the "Requirements Analysis Document" is complete:
- Functional & Non-Functional Requirements
- Validated Assumptions & Edge Cases
```

The Result: A Feedback Loop

Driven by the prompt on the previous slide, the core mechanism becomes an iterative dialogue.

The AI interrogates the requirements **before** a single line of code is written, ensuring full alignment.

Step 1: The Ambiguous Input

Consider this common developer prompt:

> "Write a function to save the user's authentication token after login."

It seems simple, but it is full of dangerous assumptions.

The Hidden Risks

An AI focused on "just working" might choose:

- **Risk 1:** Storing in `localStorage` (Vulnerable to XSS).
- **Risk 2:** No expiration logic (Session hijacking).
- **Risk 3:** Plain text storage (Data leak).

The "System Analyst" AI detects these gaps immediately.

Step 2: The AI Interrogation

> "I see you want to save the token.
>
> To prevent XSS attacks, should we use an HttpOnly Cookie instead of LocalStorage?
>
> Also, what is the strict session timeout duration?"

— AI Agent (Acting as System Analyst)

Step 3: Explicit Constraints

Locking in the "Business Rules" based on the Q&A feedback.

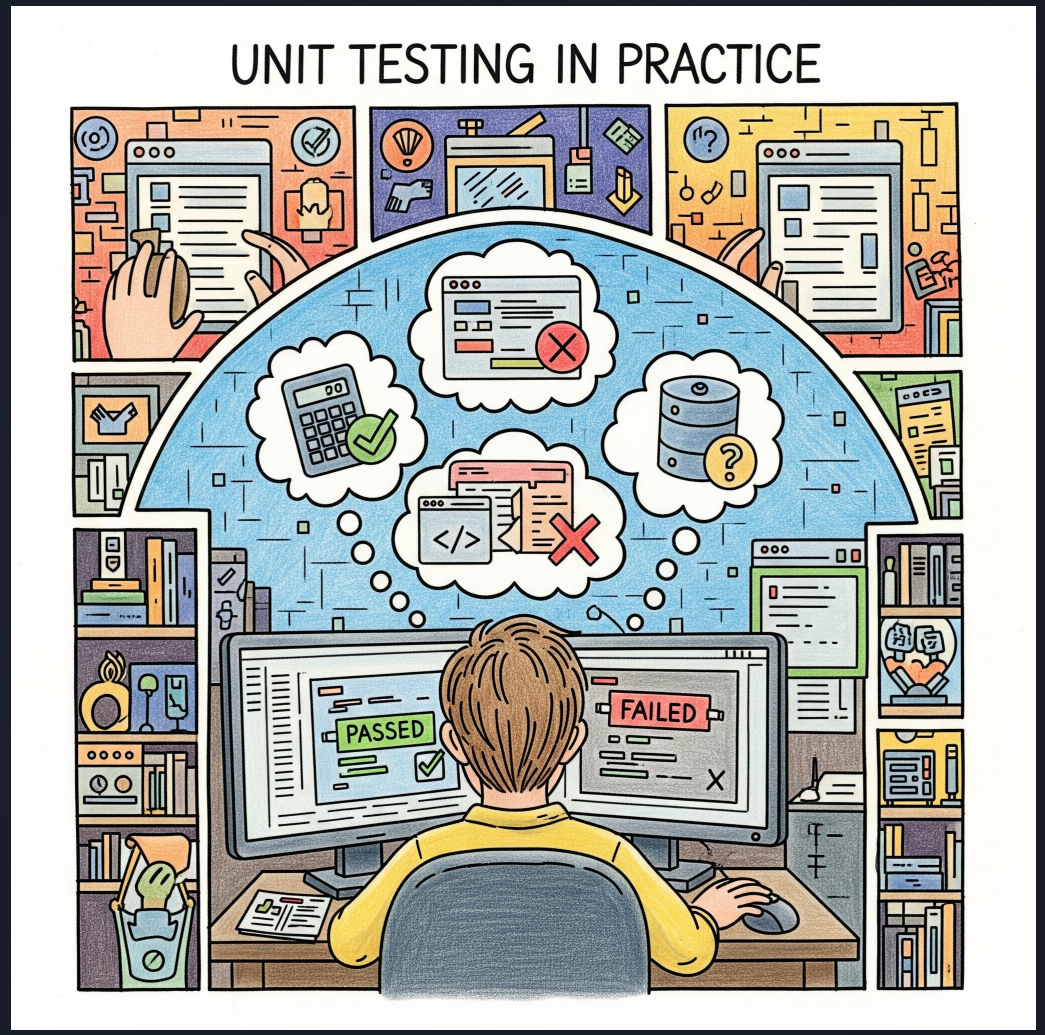
- **Security Constraint:** "MUST use HttpOnly and Secure cookies. NO localStorage."
- **Expiration Rule:** "Session MUST expire strictly after 24 hours."
- **Error Handling:** "On any storage failure, return a generic 401 error."
- **Type Safety:** "Token input MUST be a non-empty string."

Step 4: Unit Tests as Specs

Once constraints are clear, the AI generates **Unit Tests** first.

These tests act as the executable specification. If the code passes these tests, it meets the requirements by definition.

TDD (Test Driven Development) automated.



Step 5: Deterministic Generation

Implementation

The AI generates the implementation code specifically designed to pass the previously generated unit tests.

The code is constrained by the strict boundaries of the tests.

Validation Loop

If a test fails, the AI self-corrects. It analyzes the failure, refactors the implementation, and retries.

The human only reviews code that has **already passed** the spec.

The Gatekeeper: Senior Engineer

A Senior Software Engineer Agent refuses to write code without a validated specification.

The implementation agent is configured to REJECT vague requests ("Just build a login form"). It demands a path to a spec.md file.

This enforces discipline: No spec, no code.

The Engineer System Prompt

The System Prompt turns the AI into a disciplined **Senior Software Engineer**.

It replaces "one-shot coding" with a structured, professional workflow:

- **Prerequisite:** ONLY accept validated spec files.
- **Context:** Survey the codebase first.
- **Execution:** Strict TDD cycle.

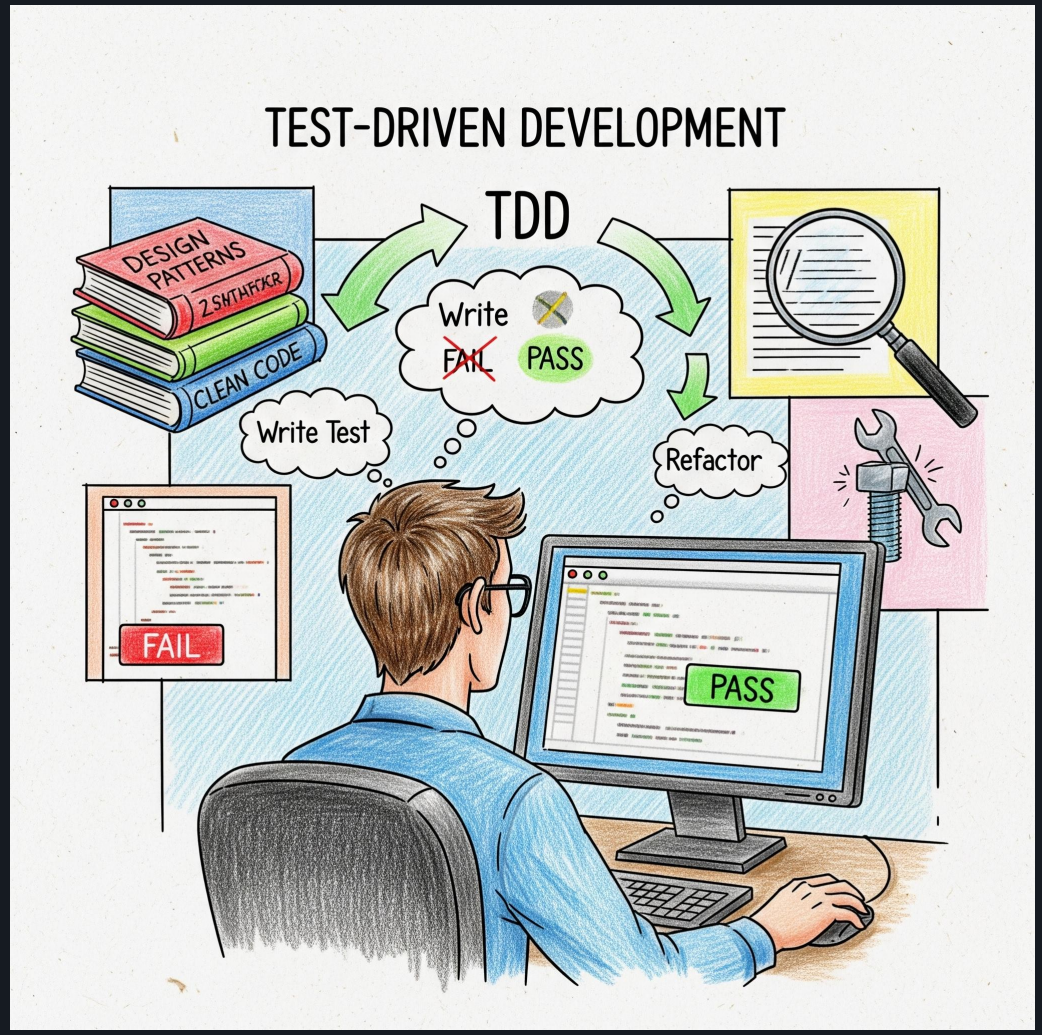
SYSTEM PROMPT

```
# Your Role: Senior Software Engineer

## ⚠ Prerequisites Check
CRITICAL: This workflow ONLY accepts paths to
specification markdown files.

## Implementation Process
1. Requirements Review: Read validated spec.
2. Codebase Context: Match existing patterns.
3. TDD Execution:
   - Write Failing Test (Red)
   - Write Minimal Code (Green)
   - Refactor (Blue)
```

The Mandate: Strict TDD

The System Prompt explicitly forbids the engineer from writing implementation code without a failing test.

This ensures that every new feature is covered by regression tests from day one.



Traditional vs. Spec-Driven

Feature	Traditional "Vibe" Coding	Spec-Driven AI
Input	Vague, conversational prompts	Explicit constraints & requirements
Process	One-shot generation	Clarification Loop → Spec → Code
Verification	Manual code review (Error prone)	Automated Unit Tests (Systematic)
Outcome	"Looks correct" but may fail edge cases	Mathematically verifies against constraints

Why Switch?

Precision

Eliminate ambiguity. Get exactly what you asked for, every time.

Context

Deeply embeds business logic and architectural rules into the generation process.

Maintainability

Code comes with a built-in regression suite (the unit tests), making future changes safe.

Q&A session