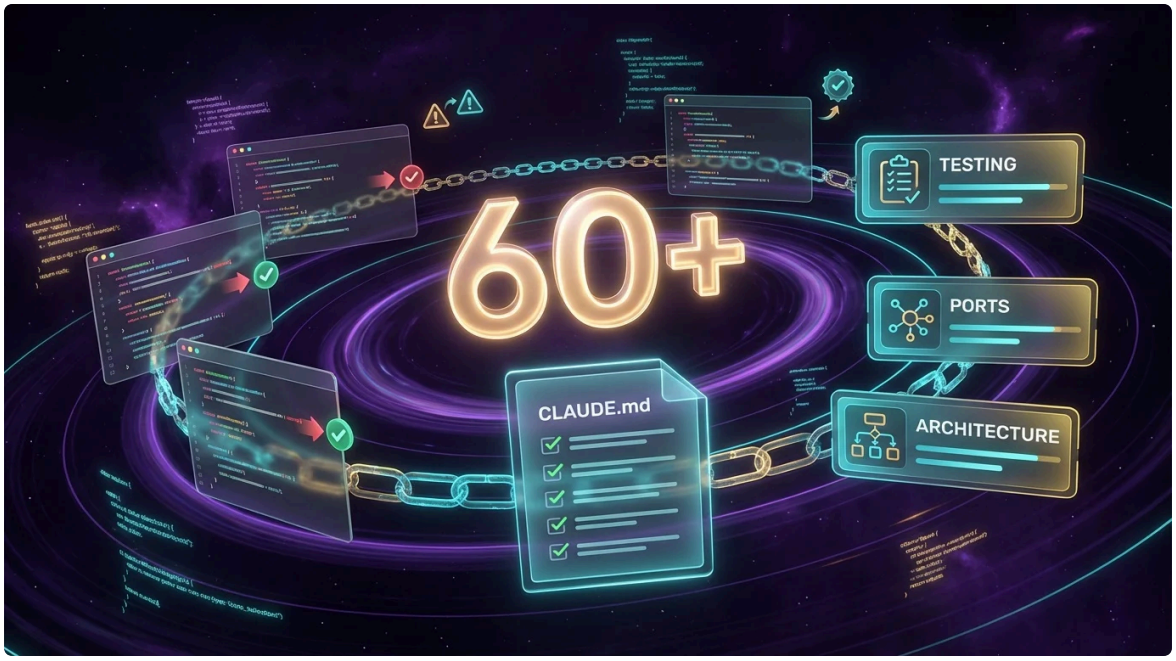


Claude Code V5 Guide

Hard-Won Lessons From 60+ Projects

FEBRUARY 11, 2026

THEDECIPHERIST.COM

TABLE OF CONTENTS

V5: HARD-WON LESSONS FROM THE TRENCHES

Previous guides: V1 | V2 | V3 | V4

V1-V4 covered the features. V5 covers what breaks — and how to prevent it. After shipping 60+ projects with Claude Code, including Classpresso (59 stars) and Wavesurf (600+ weekly downloads), I've learned the hard way what the documentation doesn't tell you.

What's new in V5:

- **Part 14: The Renaming Problem** — Why search-and-replace breaks everything
- **Part 15: Plan Mode's Hidden Flaw** — Contradictions don't auto-remove
- **Part 16: Hooks Don't Wait** — When enforcement isn't enforcement
- **Part 17: TypeScript Is Non-Negotiable** — Stop letting Claude guess
- **Part 18: Database Wrapper Pattern** — Claude loves creating connections
- **Part 19: Testing Methodology That Actually Works** — The complete framework
- **Part 20: User-Guided Testing with Claude** — Human + AI collaboration
- **Part 21: Architecture Documentation That Claude Understands** — Flowcharts and authoritative docs
- **Part 22: Fixed Ports Save Hours of Debugging** — Test mode vs prod test mode
- **Part 23: Commands as Living Documentation** — On-demand architecture diagrams
- **Part 24: Skills as Scaffolding Templates** — Consistent code generation
- **Part 25: Project Documentation That Scales** — Progress reports and infrastructure docs
- **Part 26: Workflow Hacks** — Queue mode, screenshots, and "don't rush"
- **Part 27: CLAUDE.md Rules That Save Your Ass** — The complete template

Thanks as always to the community: u/BlueVajra, u/stratofax, u/antoniocs, u/GeckoLogic, u/headset38, u/tulensrma, u/jcheroske, u/ptinsley, u/Keksy, u/lev606, and everyone who commented on V1-V4.

TL;DR: Claude Code is powerful but has specific failure modes that aren't documented. Renaming things mid-project causes bizarre side effects. Plan mode adds changes to the end without removing contradictions. Hooks execute but don't always wait. TypeScript prevents Claude from guessing types. Database wrappers prevent connection explosion. And testing isn't just "write tests" — it's a complete methodology with structured test plans, issue tracking as documentation, and human-AI collaborative testing. Plus: architecture documentation that Claude actually follows.

TABLE OF CONTENTS

FOUNDATION RECAP

V1-V4 covered the core features:

VERSION	KEY TOPICS
V1	Global CLAUDE.md, MCP servers, single-purpose chats
V2	Skills, Hooks, defense in depth
V3	LSP integration, commands/skills merge, MCP tradeoffs
V4	MCP Tool Search (85% savings), Custom Agents, Session Teleportation, Background Tasks

If you haven't read them, start there. V5 assumes you know the basics.

PART 14: THE RENAMING PROBLEM

This is probably the biggest issue I see in Claude Code. The consequences are massive.

The Scenario

You're building an npm package called `POWERLOAD`. It's working great. Then you decide:

> *"Hmm, it should be called `TURBOLOAD` instead."*

Simple rename, right? You tell Claude:

> *"Go through the project and change POWERLOAD to TURBOLOAD. Make sure all references and paths work correctly."*

Claude says "sure, no problem. DONE."

You test it. Looks OK. But then...

What Actually Happens

- Some old `.md` file still has `POWERLOAD`
- A `.txt` file was missed
- An ENV variable still references the old name
- You're in plan mode and Claude starts calling new things the old name
- Catching these issues is incredibly difficult

Why This Is Dangerous

GitHub issues confirm this is a known pain point:

> *"A recent repository-wide method renaming task took 'many hours' with Claude Code that could have been completed 'in under an hour' manually."*

Claude uses text-based search (grep, ripgrep) for renaming. It's not semantic. It doesn't understand that `POWERLOAD` in a comment, a string, a variable, and a path all need different handling.

The Solutions

1. Isolate Until Stable

If you're working on sub-modules, npm packages, or dynamic components — keep them completely isolated until you're 100% sure the name won't change. Use symlinks into your main project instead.

2. Use Environment Variables for Names

```
## In CLAUDE.md

When creating new modules or packages:
- NEVER hardcode the package/module name in code
- Use an ENV variable (e.g., PACKAGE_NAME) so it can be changed dynamically
- Reference the ENV var in package.json, imports, and documentation
```

3. If You Must Rename

If you absolutely have to rename something:

```
1. Create a checklist FIRST: all files, all references, all paths
2. Do it manually OR use your IDE's semantic rename (not Claude)
3. After renaming, do a full project search for the old name
4. Check: .md files, .txt files, .env files, comments, strings, paths
5. Start a fresh Claude session after renaming
```

4. There's a Tool for This

curiouslychase created `claude-mv` specifically for renaming projects without losing Claude context. If you rename a directory, Claude's conversations get orphaned in `~/.claude/` under the old path.

PART 15: PLAN MODE'S HIDDEN FLAW

Always use plan mode for bigger changes. Verify the plan before saying yes. But be aware of something important.

The Problem

Just because you expect Claude to accept your changes doesn't mean Claude will automatically go back through the plan and remove contradictions.

What I Encountered

I asked Claude to fundamentally change something in the plan. Here's what happened:

1. Claude added my change **to the end of the plan**
2. Claude did NOT go back and remove the contradicting steps earlier in the plan
3. I approved the plan, not realizing the contradiction
4. Claude executed both the old approach AND my new approach
5. Chaos ensued

I asked Claude what I should have done. Claude admitted:

> *"I should have caught this. But I didn't."*

The GitHub Issues Confirm This

Multiple bugs reported:

> *"Claude isn't adhering to plan mode. It quite often 'breaks out' and starts modifying files."*

Another issue:

> *"When implementing a plan, Claude Code ignored the main 'ALWAYS' rules in my CLAUDE.md."*

And another:

> *"Claude interpreted the 'Exited Plan Mode' message as authorization to proceed with edits, even though the user never explicitly approved."*

The Solutions

1. Name Your Steps Consistently

When modifying a plan, use the EXACT same step names. This helps Claude understand what you're replacing, not appending.

```
❌ Bad: "Also, use Redis instead"
✅ Good: "Replace Step 3 (Database Selection): Use Redis instead of MongoDB"
```

2. Ask Claude to Review for Contradictions

After making a plan change:

```
Before I approve this plan, review it for contradictions.
Are there any steps that conflict with each other?
List them.
```

3. Don't Just Skim

I know. Reading the same plan over and over is tedious. But if you modified something, you MUST verify Claude removed the old approach.

4. Use /clear Between Major Plan Changes

If you're fundamentally changing direction, it might be faster to:

1. `/clear`
2. State your new requirements fresh
3. Generate a new plan from scratch

PART 16: HOOKS DON'T WAIT

Here's something the documentation doesn't make clear: hooks execute, but they don't always wait.

The Official Behavior

From the Claude Code docs:

> *"When an async hook fires, Claude Code starts the hook process and immediately continues without waiting for it to finish."*

For **async hooks**, this is documented. But even for sync hooks, I've noticed Claude sometimes says "done waiting" if a shell script is too slow.

Known Issues

GitHub is full of hook problems:

- Hooks not executing despite proper configuration
- PreToolUse hooks running but not blocking operations
- Exit codes not being respected

The Solutions

1. Keep Hooks Fast

If a hook takes more than ~5 seconds, Claude may timeout waiting. Keep your hooks lightweight.

2. Use Exit Code 2 to Block

From the docs: exit code 2 blocks the operation. But test this independently – it doesn't always work as documented.

3. Test Hooks Outside Claude

Before trusting a hook, run it manually:

```
bash -x ~/.claude/hooks/your-hook.sh
echo $?
```

4. The `async: true` Option

As of January 2026, hooks support an `async: true` option for background operations that don't need to block:

```
{
  "hooks": {
    "PostToolUse": [
      {
        "command": "node ~/.claude/hooks/log-tool-use.js",
        "async": true
      }
    ]
  }
}
```

PART 17: TYPESCRIPT IS NON-NEGOTIABLE

This one is simple but crucial.

The Problem

Claude guesses types in JavaScript. It doesn't know your function returns `User | null` vs `User`. It doesn't know your array is `string[]` vs `number[]`. It guesses.

And guesses become runtime errors.

The Evidence

Builder.io's analysis:

> *"AI tools work significantly better with TypeScript. Types provide explicit context that AI can use."*

ClaudeLog reports:

> *"JavaScript & TypeScript show the strongest community satisfaction when paired with Claude Code."*

Internal benchmarks show ~88% accuracy on TypeScript refactoring tasks vs significantly lower for JavaScript where Claude has to infer types.

The Solution

In your CLAUDE.md:

```
## TypeScript

- ALWAYS use TypeScript for new files
- ALWAYS use strict mode
- NEVER use `any` unless absolutely necessary
- When editing JavaScript files, convert to TypeScript first
```

Why This Matters

Claude reads type definitions like specs. When your function signature says:

```
function getUser(id: string): Promise<User | null>
```

Claude knows:

- Input is a string (not number)
- Output is async
- Output can be null (must handle)

Without types, Claude guesses all of this.

PART 18: THE DATABASE WRAPPER PATTERN

This one cost me hours of debugging.

The Problem

Claude loves creating new database connections. Every file that needs data, Claude adds:

```
const client = new MongoClient(process.env.MONGODB_URI);
await client.connect();
```

The result: **connection pool explosion**. You hit MongoDB's connection limit. Things start failing randomly.

The Solution

Create a centralized database wrapper:

```
project/
├── core/
│   └── db/
│       └── db.js    # THE ONLY FILE THAT CONNECTS
├── services/
│   ├── users.js     # imports from core/db/db.js
│   └── orders.js    # imports from core/db/db.js
```

db.js:

```
// Singleton connection
let client = null;
let db = null;

async function connect() {
  if (db) return db;
  client = new MongoClient(process.env.MONGODB_URI);
  await client.connect();
  db = client.db('myapp');
  return db;
}

export async function getData(collection, query) {
  const database = await connect();
  return database.collection(collection).find(query).toArray();
}

export async function saveData(collection, data) {
  const database = await connect();
  return database.collection(collection).insertOne(data);
}
```

The CLAUDE.md Rule

```
## Database

- NEVER directly establish database connections outside of core/db/
- ALWAYS use getData() and saveData() from core/db/db.js
- NEVER create new MongoClient instances
- NEVER call client.connect() directly
```

Why This Pattern Works

1. **Single connection pool** – No explosion
2. **Single point of change** – Update MongoDB driver in one place
3. **Testable** – Mock the wrapper, not the driver
4. **Loggable** – Add logging in one place

PART 19: TESTING METHODOLOGY THAT ACTUALLY WORKS

"Write tests" is not a methodology. Here's what actually works after 60+ projects.

The Test MD Template

Every test file should follow this structure:

````
# [Feature] Test

**Last Updated:** [date]
**Purpose:** [one line description]

---

## Quick Status

```
Feature A:        ✅ PASSED
Feature B:        ⬜ NOT TESTED
Feature C:        ❌ FAILED
```

---

## Prerequisites

- [ ] Service X running on port Y
- [ ] Test user created
- [ ] Environment variables set

---

## Test 1: [Specific Scenario]

### 1.1 [Sub-test Name]

**Action:** [What to do]
**Expected:** [What should happen]

| Check | Expected | Actual | Status |
|-------|----------|--------|--------|
| Response code | 200 | | ⬜ |
| Data returned | User object | | ⬜ |

### 1.2 [Another Sub-test]

**Action:** [What to do]
```bash
curl -X POST https://api.example.com/endpoint \
  -H "Content-Type: application/json" \
  -d '{"key": "value"}'
```
**Expected:**
````

```
```json
{
  "status": "success"
}
```

Pass/Fail Criteria

Criteria	Pass	Fail
All endpoints respond	Yes	Any 500
Data persists	Yes	Lost on refresh

Sign-Off

Test	Tester	Date	Status
Test 1			⬜
Test 2			⬜
```

Test Types You Need

TYPE	PURPOSE	WHEN TO USE
Plan Enforcement	Verify tier gating	SaaS features
Feature Flow	Complete user journey	New features
Compliance	GDPR, security	Legal requirements
Infrastructure	Scaling, load	Pre-launch
User-Guided	Exploratory	UI polish
E2E Log	Document test runs	Critical paths

The Master Checklist Pattern

Create a `CHECKLIST.md` as your single source of truth:

````
# Test Checklist

**Last Updated:** 2026-02-03

---

## Authentication

| Test | Status | Notes |
|------|--------|-------|
| Login | ✅ | Tested via Playwright |
| Signup | ✅ | E2E tested |
| Password reset | ⬜ | Needs testing |

---

## Billing

| Test | Status | Notes |
|------|--------|-------|
| Stripe checkout | ✅ | Test mode verified |
| Webhook delivery | ✅ | Both regions |
| Trial expiration | ⬜ | Need test clocks |

---

## Quick Status

```
Authentication:      ✅ 2/3 PASSED
Billing:             ✅ 2/3 PASSED
Dashboard UI:        ⬜ NOT STARTED
```
````

Phase-Based Testing

Don't test everything at once. Break it down:

Phase 1: Unit Tests (Vitest)

- After every component/function
- Run with: `npm test`
- Focus: Individual units work

Phase 2: Integration Tests

- After connecting services
- Focus: Services talk to each other

Phase 3: E2E Tests (Playwright)

- After feature completion
- Focus: User flows work end-to-end

The Success Criteria Problem

Claude's E2E tests often pass when they shouldn't:

```
// ❌ BAD: Passes even if broken
test('user can login', async ({ page }) => {
  await page.goto('/login');
  await page.fill('#email', 'test@example.com');
  await page.fill('#password', 'password');
  await page.click('button[type="submit"]');
  // No assertion! This always passes!
});
```

```
// ✅ GOOD: Explicit success criteria
test('user can login', async ({ page }) => {
  await page.goto('/login');
  await page.fill('#email', 'test@example.com');
  await page.fill('#password', 'password');
  await page.click('button[type="submit"]');

  // Explicit success criteria
  await expect(page).toHaveURL('/dashboard');
  await expect(page.locator('h1')).toContainText('Welcome');
  await expect(page.locator('[data-testid="user-email"]')).toContainText('test@
});
```

In your CLAUDE.md:

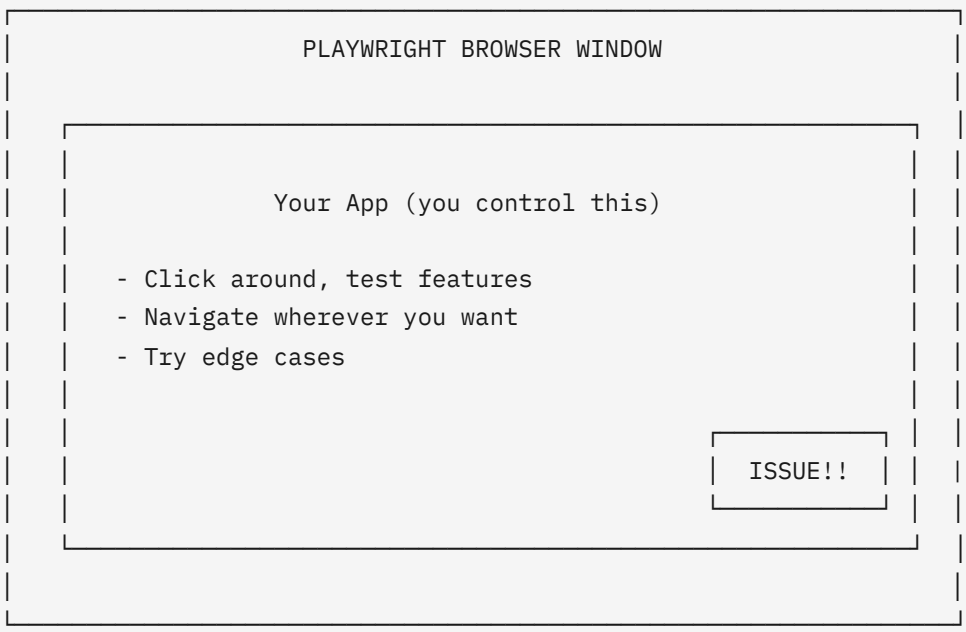
```
## Testing

- ALWAYS define explicit success criteria for E2E tests
- "Page loads" is NOT a success criterion
- Every test must verify: URL, visible elements, data displayed
```

PART 20: USER-GUIDED TESTING WITH CLAUDE

Here's a testing pattern I developed that combines human intuition with Claude's fixing ability.

The Concept



How It Works

Step 1: Tell Claude: "Start user-guided test on dashboard"

Claude opens Playwright, injects a floating "ISSUE" button, and gives you control.

Step 2: You explore freely — clicking around, testing edge cases, checking responsive layouts.

Step 3: When you find a problem, click the red "ISSUE" button:

- Screenshot is automatically captured
- Modal opens for your description
- Issue is appended to `ISSUES_FOUND.md`

Step 4: When done, say: "SOLVE ISSUES_FOUND.md"

Claude reads all issues, analyzes screenshots, finds the code, and fixes each one systematically.

The ISSUES_FOUND.md Format

```
# Issues Found During User-Guided Testing

**Test Session:** 2026-02-05
**Tester:** You (manual) + Claude
**Environment:** http://localhost:3000

---

## Issue #1

**Timestamp:** 2026-02-05T14:35:22Z
**URL:** /dashboard/alerts
**Screenshot:** ./screenshots/issue-001.png

### Description
The "Create Alert" button is barely visible - text color blends with backgrounc
Expected: Clear, readable button
Actual: Very low contrast, hard to see

### Status
⬜ Pending

---

## Issue #2

**Timestamp:** 2026-02-05T14:38:45Z
**URL:** /dashboard/rules
**Screenshot:** ./screenshots/issue-002.png

### Description
Custom rule form doesn't show validation errors. I submitted empty fields
and nothing happened - no error message, no indication of what's wrong.

### Status
⬜ Pending
```

After "SOLVE ISSUES_FOUND.md"

Claude updates each issue with resolution:

```
## Issue #1

**Status:** ✅ Resolved

### Resolution
Fixed button contrast in `/src/app/dashboard/alerts/page.tsx`
- Changed `text-text-muted` to `text-text-primary`
- Added `hover:bg-accent-green/90` for better feedback

### Root Cause
Dark theme color tokens were using low-contrast combination.
```

Why This Works

1. **Human intuition** finds problems automated tests miss
2. **Screenshots** give Claude full context
3. **Structured format** makes fixes systematic
4. **Issues become documentation** – you can see what was fixed

The ISSUES_FIXED.md Archive

Keep a permanent record of all fixes:

```
# Issues Fixed During Testing

**Test Session:** 2026-02-05

---

## Fix #1 - NEXTAUTH_URL Wrong Port

**Issue**: Redirecting to `localhost:3000` instead of `localhost:3002`
**File**: `.env.local`
**Change**: `NEXTAUTH_URL=http://localhost:3002`
**Root Cause**: Copy-paste error from another project

---

## Fix #2 - Rate Limit Key Invalid

**Issue**: "Cannot read properties of undefined"
**File**: `src/app/api/v1/user/save-onboarding/route.ts`
**Change**: `rateLimit(request, 'general')` → `rateLimit(request, 'api')`
**Root Cause**: 'general' is not a valid rate limit key
```

This pattern turns bugs into documentation. Over time, you build a knowledge base of what went wrong and how you fixed it.

PART 21: ARCHITECTURE DOCUMENTATION THAT CLAUDE UNDERSTANDS

Claude reads your docs. Make sure they're worth reading.

The Flowchart Pattern

ASCII architecture diagrams work surprisingly well:

````
# Application Flowchart

## Data Flow

```
┌─────────────┐    events    ┌─────────────┐
│   Client    │─────────────>│     API     │
│   (hooks)   │              │  (Express)  │
└─────────────┘              └──────┬──────┘
                                    │
                                    │ WRITE
                                    ▼
                             ┌─────────────┐
                             │   MongoDB   │
                             └──────┬──────┘
                                    │
                    ┌───────────────┼──────────────┐
                    │               │              │
                    ▼ WATCH         ▼ READ         │
             ┌─────────────┐  ┌─────────────┐       │
             │    Tasks    │  │  Dashboard  │       │
             │(ChangeStream)│  │  (Next.js)  │       │
             └──────┬──────┘  └─────────────┘       │
                    │                              │
                    │ WRITE                        │
                    ▼                              │
             ┌─────────────┐                       │
             │ violations  │───────────────────────┘
             │ collection  │        READ
             └─────────────┘
```
````

Claude reads this and understands:

- What writes to what
- What reads from what
- The flow of data through your system

The Authoritative Document Pattern

Some documents should be **absolute constraints**:

```
# Service Separation Logic

> **This document is AUTHORITATIVE. No exceptions. No deviations.**

---

## Traffic Rules (IRONCLAD)

### Client → API ONLY
```
┌──────────────┐   ┌──────────────────────┐
│   Client   │─────────>│   api.example.com   │
│   │   │   │
└──────────────┘   │ POST /api/v1/   │
                          │   - ingest   │
       NEVER   │   - session   │
         ▼   │   - heartbeat   │
┌──────────────┐   └──────────────────────┘
│ Dashboard   │
│ (BLOCKED)   │
└──────────────┘
```

### If You Are About To...

If you are about to:
- Add an ingest endpoint to Dashboard → **STOP. Add it to API.**
- Add a billing endpoint to API → **STOP. Add it to Dashboard.**
- Make Client call Dashboard → **STOP. It calls API.**

**This document overrides all other instructions.**
```

Why This Works

The key phrases:

- "This document is AUTHORITATIVE"
- "No exceptions. No deviations"
- "IRONCLAD"
- "If you are about to... STOP"
- "This document overrides all other instructions"

These tell Claude this isn't a suggestion – it's a constraint.

The E2E Test Log Pattern

Document complete test runs:

````
# End-to-End Test Log

**Started:** 2026-02-03 21:11 UTC
**Completed:** 2026-02-03 21:40 UTC
**Status:** ✅ ALL TESTS PASSED (with 1 bug fix deployed)

---

## Phase 1: Authentication

| Step | Action | Result | Status |
|------|--------|--------|--------|
| 1.1 | Navigate to /signup | Page loaded | ✅ |
| 1.2 | Complete registration | Account created | ✅ |
| 1.3 | Retrieve API key | Key obtained | ✅ |

---

## BUG DISCOVERED AND FIXED

### Issue
Trial showed as expired immediately after signup.

### Root Cause
`trialEndsAt` was not being saved when creating subscription.

### Fix Applied
Added `trialEndsAt` calculation to subscription creation:
```typescript
const trialEndsAt = subscription.trial_end
  ? new Date(subscription.trial_end * 1000)
  : new Date(Date.now() + 7 * 24 * 60 * 60 * 1000);
```

### Fix Verification
Created new user after fix — now shows "Trial: 7 days remaining"

**Status:** ✅ FIX DEPLOYED AND VERIFIED

---

## Summary

| Category | Passed | Total |
|----------|--------|-------|
| Authentication | 4 | 4 |
````

```
| API Endpoints | 8 | 8 |
| Dashboard UI | 6 | 6 |

**End-to-End Test: PASSED** ✅
```

This becomes a **reference document** for future debugging. When something breaks, you can trace back to when it last worked.

Reference Documentation Structure

For larger projects, break documentation into focused files:

```
docs/
├── ARCHITECTURE.md      # System overview
├── DATA_FLOW.md         # How data moves
├── API_CONTRACTS.md     # API specifications
├── SERVICE_A.md         # Service A details
├── SERVICE_B.md         # Service B details
└── SEPARATION.md        # What goes where (authoritative)
```

Then in CLAUDE.md:

```
## Project Documentation

When you need to understand:
- System architecture: Read `./docs/ARCHITECTURE.md`
- Data flow: Read `./docs/DATA_FLOW.md`
- What goes where: Read `./docs/SEPARATION.md` (AUTHORITATIVE)

ALWAYS read relevant docs before making changes that span services.
```

PART 22: FIXED PORTS SAVE HOURS OF DEBUGGING

This seems trivial until you've lost an afternoon to it.

The Problem

Claude loves port 3000. Every new service, every new project – Claude defaults to 3000. When you're working on a larger application with multiple services, or especially when you have **multiple VS Code windows open at the same time**, chaos ensues:

- Service A starts on 3000
- Service B tries to start on 3000 → port already in use
- Claude doesn't notice, keeps coding against the wrong service
- You're testing changes that aren't being served
- Half an hour later you realize nothing is working

Claude half the time doesn't know if it's working on the correct thing at all. It sees "server running" and assumes everything is fine.

The Solution

Define fixed ports for EVERY service in EVERY project. Put it in CLAUDE.md.

````
## Service Ports (FIXED - NEVER CHANGE)

| Service | Port | URL |
|---------|------|-----|
| Website | 3000 | http://localhost:3000 |
| API | 3001 | http://localhost:3001 |
| Dashboard | 3002 | http://localhost:3002 |
| Admin Tools | 3003 | http://localhost:3003 |
| Analytics | 3021 | http://localhost:3021 |
| Tasks | – | (no HTTP server) |

**Port 3002 is ALWAYS the customer Dashboard. Port 3003 is ALWAYS Admin Tools.

When starting any service, ALWAYS use its assigned port:
```bash
# CORRECT
npx next dev -p 3002

# WRONG - never let it default
npx next dev
```
````

Test Mode vs Prod Test Mode

When you need to test against local services AND production services simultaneously (or switch between them without conflicts), use **port ranges**:

````
## Port Modes

### TEST MODE (Local Development)
When I say "start test mode", use these ports:

| Service | Port | URL |
|---------|------|-----|
| Website | 4000 | http://localhost:4000 |
| API | 4010 | http://localhost:4010 |
| Dashboard | 4020 | http://localhost:4020 |
| Admin | 4030 | http://localhost:4030 |
| Analytics | 4040 | http://localhost:4040 |

### PROD TEST MODE (Testing Against Production)
When I say "start prod test mode", use these ports:

| Service | Port | URL |
|---------|------|-----|
| Website | 5000 | http://localhost:5000 |
| API | 5010 | http://localhost:5010 |
| Dashboard | 5020 | http://localhost:5020 |
| Admin | 5030 | http://localhost:5030 |
| Analytics | 5040 | http://localhost:5040 |

### ⚠️ CRITICAL: ALWAYS CLOSE RUNNING SERVICES FIRST

**BEFORE starting any mode, ALWAYS kill services on the related ports:**

```bash
# Kill TEST MODE ports
lsof -ti:4000,4010,4020,4030,4040 | xargs kill -9 2>/dev/null

# Kill PROD TEST MODE ports
lsof -ti:5000,5010,5020,5030,5040 | xargs kill -9 2>/dev/null
```

**If you don't kill existing processes first:**
- New service silently fails to bind
- Old code keeps running
- You test changes that don't exist
- Hours of confusion
````

Why Port Ranges?

RANGE	PURPOSE	WHEN TO USE
3000-3099	Production-like	CI/CD, final testing
4000-4099	Test mode	Active development
5000-5099	Prod test mode	Testing against real APIs

The 10-increment pattern (4000, 4010, 4020) leaves room for related services:

- 4020 = Dashboard
- 4021 = Dashboard worker (if needed)
- 4022 = Dashboard websocket (if needed)

The Start Command Template

````
## Start Commands

### Test Mode
```bash
# ALWAYS kill first
lsof -ti:4000,4010,4020,4030,4040 | xargs kill -9 2>/dev/null

# Then start
PORT=4000 pnpm --filter website dev &
PORT=4010 pnpm --filter api dev &
PORT=4020 pnpm --filter dashboard dev &
PORT=4030 NEXT_PUBLIC_APP_MODE=admin pnpm --filter dashboard dev &
PORT=4040 pnpm --filter analytics dev &
```

### Prod Test Mode
```bash
# ALWAYS kill first
lsof -ti:5000,5010,5020,5030,5040 | xargs kill -9 2>/dev/null

# Then start (pointing to production APIs)
PORT=5000 NEXT_PUBLIC_API_URL=https://api.example.com pnpm --filter website dev
PORT=5010 pnpm --filter api dev &
PORT=5020 NEXT_PUBLIC_API_URL=https://api.example.com pnpm --filter dashboard d
# ... etc
```
````

Why Specific Port Assignments Matter

1. **Muscle memory** – You always know Dashboard is 4020 in test mode
2. **Multiple VS Codes** – Each service has its own port, no conflicts
3. **Mode switching** – Jump between test and prod test without killing everything
4. **Bookmarks work** – Your browser bookmarks stay valid per mode
5. **Claude knows where to look** – "Check localhost:4020" is unambiguous
6. **Env vars stay consistent** – `NEXT_PUBLIC_API_URL=http://localhost:4010` never changes within a mode

The "What's Running Where" Check

When debugging, tell Claude:

```
Before making any changes, confirm:
1. Which mode are we in? (test / prod test)
2. Which service handles this endpoint?
3. What port is it running on in this mode?
4. Is that service actually running right now?
```

This catches the "Claude is editing the API but testing against the Dashboard" problem.

The Real Customer Test

Before launching, test the **entire customer journey** — not just individual features. Write it as a test plan:

```
# Real Customer Test

**CRITICAL: Use production environment (or staging). Not localhost.**

This test simulates a real customer going through your entire flow.

## Phase 1: Installation & Signup

1. Download and install the NPM package
   a. Create new test account (fresh email, not your dev account)
   b. Follow the actual installation instructions from the README
   c. Go to dashboard and register as this new user
   d. Choose a paid plan and complete payment
   e. Verify the CLI/package is now sending data correctly
   f. Monitor the API logs to confirm data is received
   g. Verify the API key works

2. Checkpoint: Data is flowing from customer machine → API → Dashboard

## Phase 2: Core Feature Test

1. Create a test project: `~/projects/customer_test`
2. Open your tool from that directory
3. Do something that should trigger your product's features
   - Example: "build me a modern fitness website with SSR"
4. Use Playwright against production dashboard to verify:
   - Data appears correctly
   - Features work as expected
   - No errors in UI

## Phase 3: Advanced Features

1. Install any additional tools (MCP server, CLI extensions, etc.)
2. Test each major command/feature:
   - Command 1: Expected result?
   - Command 2: Expected result?
   - Command 3: Expected result?
3. Verify outputs match expectations
4. Execute any suggested actions
5. Verify the actions had the expected effect

## Phase 4: Edge Cases

1. What happens with no data?
2. What happens with bad credentials?
3. What happens when services are down?
```

```
## Sign-Off

| Phase | Status | Notes |
|-------|--------|-------|
| Installation & Signup | ⬜ | |
| Core Feature | ⬜ | |
| Advanced Features | ⬜ | |
| Edge Cases | ⬜ | |
```

Why this matters:

- Unit tests pass but the product doesn't work
- Integration tests pass but the install flow is broken
- Everything works locally but fails in production

The Real Customer Test catches what other tests miss because it uses **production infrastructure** with a **fresh account** following **actual documentation**.

PART 23: COMMANDS AS LIVING DOCUMENTATION

Commands aren't just shortcuts. They're **reference documents** that Claude can display on demand.

The Problem

You have complex system architecture. You explain it to Claude. Next session, Claude forgets. You explain again. And again.

The Solution: Documentation Commands

Create commands that display your architecture:

```
# ~/.claude/commands/connections.md

---
description: Show system connection flowchart
---

# System Connections

Display the current connection status.

## Flowchart

```
┌──────────────────────────────────────────────────────────────────┐
│  LOCAL (Developer's Machine)                                     │
│                                                                  │
│    ┌──────────────┐                          ┌──────────────┐   │
│    │  CLAUDE AI   │                          │  DASHBOARD   │   │
│    │              │                          │   :3000      │   │
│    └──────┬───────┘                          └──────┬───────┘   │
│           │                                         │           │
│           │ MCP (stdio)                      HTTP   │           │
│           │                                         │           │
│           ▼                                         ▼           │
│    ┌──────────────┐                          ┌──────────────┐   │
│    │   Captain    │                          │   Bridge     │   │
│    │   (MCP)      │                          │   :3100      │   │
│    └──────┬───────┘                          └──────────────┘   │
│           │                                                     │
│     ┌─────┼─────┐                                               │
│     ▼     ▼     ▼                                               │
│   [DB]  [API]  [Auth]                                           │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘
```

Run `/connections` anytime to see current system status.
```

When you type `/connections`, Claude displays your entire architecture. Every session. No re-explaining.

The Progress Command Pattern

A command that checks actual state and reports status:

```
# ~/.claude/commands/progress.md

---
description: Show project progress - what's done, what's pending
---

# Project Progress

Check the status of all components.

## Instructions

1. Read `PROJECT_STATUS.md` for context
2. Check each folder in `~/projects/myapp/packages/`
3. For each package, determine:
   - Does `src/index.ts` exist? (Has Code)
   - Does `dist/` folder exist? (Built)
4. Output results table

## Shell Commands to Run

```bash
for pkg in ~/projects/myapp/packages/*; do
  name=$(basename "$pkg")
  has_code=$( [ -f "$pkg/src/index.ts" ] && echo "✅" || echo "❌" )
  has_dist=$( [ -d "$pkg/dist" ] && echo "✅" || echo "❌" )
  echo "$name|$has_code|$has_dist"
done
```

Output Format

Package	Has Code	Built	Status
api	✅	✅	COMPLETE
dashboard	✅	❌	PARTIAL
cli	❌	❌	NOT STARTED
```

Now `/progress` gives you a real-time status report based on actual filesystem state.

The Fleet Roster Pattern

For complex systems with many components:

```
# ~/.claude/commands/fleet.md

---
description: Show all services and their roles
---

# Service Fleet

## Summary

| Stat | Count |
|------|-------|
| Total Services | 12 |
| Running | 8 |
| Pending | 4 |

## Roster

| Service | Domain | Role | Status |
|---------|--------|------|--------|
| Website | example.com | Marketing | ✅ |
| API | api.example.com | Data ingestion | ✅ |
| Dashboard | dashboard.example.com | Customer UI | ✅ |
| Tasks | (internal) | Background jobs | ✅ |
```

This becomes your **single source of truth** for what exists and what it does.

Why Commands Beat CLAUDE.md for Reference

APPROACH	PROBLEM
Put it in CLAUDE.md	Bloats context, always loaded
Explain each session	Wastes time, inconsistent
Use commands	**On-demand, consistent, versioned**

Commands are loaded only when called. Your architecture diagram doesn't consume tokens until you need it.

PART 24: SKILLS AS SCAFFOLDING TEMPLATES

Skills aren't just instructions. They're **complete scaffolding systems** that generate consistent code.

The Create Command Pattern

Instead of explaining how to create a new component every time:

````
# ~/.claude/skills/create-service/SKILL.md

---
description: Create a new microservice that follows our patterns
---

# Create Service Skill

Generate a new service that works with our architecture.

## Architecture

```
┌─────────────────────────────────────────────────────────────┐
│                      YOUR SERVICE                           │
├─────────────────────────────────────────────────────────────┤
│  ┌─────────────────────────────────────────────────────┐    │
│  │              SERVER (server.ts)                     │    │
│  │  - Express/Fastify entry point                      │    │
│  │  - Defines routes                                   │    │
│  │  - NEVER contains business logic                    │    │
│  └─────────────────────────────────────────────────────┘    │
│                           │                                 │
│                           ▼                                 │
│  ┌─────────────────────────────────────────────────────┐    │
│  │                 HANDLERS (handlers/)                │    │
│  │  - Business logic lives here                        │    │
│  │  - One file per domain                              │    │
│  └─────────────────────────────────────────────────────┘    │
│                           │                                 │
│                           ▼                                 │
│  ┌─────────────────────────────────────────────────────┐    │
│  │                 ADAPTERS (adapters/)                │    │
│  │  - External service wrappers                        │    │
│  │  - Database, APIs, etc.                             │    │
│  └─────────────────────────────────────────────────────┘    │
└─────────────────────────────────────────────────────────────┘
```

## Directory Structure

```
packages/{name}/
├── src/
│   ├── server.ts        # Entry point
│   ├── handlers/        # Business logic
````
````

```
│   │   └── index.ts
│   ├── adapters/        # External services
│   │   └── index.ts
│   └── types.ts         # TypeScript types
├── package.json
├── tsconfig.json
├── Dockerfile
└── CLAUDE.md
```

Template: package.json

```json
{
  "name": "@myapp/{name}",
  "version": "1.0.0",
  "type": "module",
  "scripts": {
    "build": "tsc",
    "dev": "tsx watch src/server.ts",
    "start": "node dist/server.js"
  }
}
```

Template: src/server.ts

```typescript
import express from 'express';
import { handlers } from './handlers/index.js';

const app = express();
const PORT = process.env.PORT || 3000;

app.use(express.json());

// Routes - delegate to handlers
app.get('/health', handlers.health);
app.post('/api/v1/:action', handlers.handleAction);

app.listen(PORT, () => {
  console.log(`{name} running on port ${PORT}`);
});
```

Template: Dockerfile

```dockerfile
FROM node:20-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY dist ./dist
CMD ["node", "dist/server.js"]
```

Category Interfaces

New services must match existing patterns:

API Services
```
/health - Health check
/api/v1/* - Versioned endpoints
```

Background Services
```
No HTTP server
Connects to message queue or change streams
```

After Creating

1. Install: `pnpm install`
2. Build: `pnpm build`
3. Test: `pnpm dev`
4. Add to docker-compose.yml
5. Add to CHECKLIST.md

Checklist

- [ ] Directory structure matches template
- [ ] package.json has correct scripts
- [ ] Dockerfile follows pattern
- [ ] CLAUDE.md documents the service
- [ ] Added to docker-compose.yml
- [ ] Added to CHECKLIST.md
- [ ] Port assigned in CLAUDE.md port table

Why Skills Beat Ad-Hoc Instructions

WITHOUT SKILLS	WITH SKILLS
"Create a new service" → Claude invents structure	"Create a new service" → Claude follows template
Inconsistent patterns across services	Every service looks the same
Forget Dockerfile, forget types, forget tests	Checklist ensures completeness
Re-explain architecture every time	Architecture diagram in skill

Skill Categories

SKILL	PURPOSE
`create-service`	New microservice scaffolding
`create-component`	React component with tests
`create-api-route`	API endpoint with validation
`create-test`	Test file with proper structure
`create-migration`	Database migration with rollback

Each skill is a **complete template** — not just instructions, but actual file contents that Claude copies and customizes.

PART 25: PROJECT DOCUMENTATION THAT SCALES

As projects grow, you need more than CLAUDE.md. Here are documentation patterns that keep Claude aligned across large codebases.

The Progress Report Pattern

Track project status in a living document:

```
# Project Progress Report

> Last Updated: 2026-02-11 (Session 47)

---

## Phase Summary

| Phase | Description | Status | Progress |
|-------|-------------|--------|----------|
| Phase 1 | Core Infrastructure | **Complete** | 100% |
| Phase 2 | API Layer | In Progress | ~75% |
| Phase 3 | Dashboard UI | Not Started | 0% |

---

## Phase 2: API Layer

### 2.1 Authentication

| Component | Has Code | Built | Tested | Status |
|-----------|----------|-------|--------|--------|
| JWT auth | ✅ | ✅ | ✅ | **COMPLETE** |
| OAuth | ✅ | ✅ | ❌ | **PARTIAL** |
| API keys | ❌ | ❌ | ❌ | **NOT STARTED** |

**TODO:**
- [x] Implement JWT authentication
- [x] Build OAuth flow
- [ ] Test OAuth with real providers
- [ ] Create API key management

**Notes:**
- OAuth uses PKCE flow for security
- API keys will support scopes and rate limits

---

## Session Changelog

### 2026-02-11 (Session 47)
- Completed JWT authentication
- Started OAuth implementation
- Updated Phase 2 progress to 75%

### 2026-02-10 (Session 46)
```

```
- Fixed database connection pooling
- Added rate limiting middleware

---

## Next Actions (Priority Order)

1. **Test OAuth** - Verify Google/GitHub providers work
2. **API Keys** - Implement key generation and validation
3. **Rate Limiting** - Add per-key rate limits
```

Why this works:

- Phase summary shows big picture at a glance
- Component tables show granular status
- Session changelog tracks what changed when
- Next Actions keeps priorities clear

The Infrastructure Document Pattern

For complex systems, create a single source of truth:

````
# Infrastructure

> **Always check this file before making programming decisions.**

---

## Architecture Overview

```
┌─────────────────────────────────────────────────────────────────┐
│                      CLOUD (api.example.com)                    │
│   • User accounts & auth                                        │
│   • Billing & subscriptions                                     │
│   • License management                                          │
└─────────────────────────────────────────────────────────────────┘
                              ↑
                         auth / license
                              ↓
┌─────────────────────────────────────────────────────────────────┐
│                    CUSTOMER'S MACHINE                           │
│   • Dashboard (localhost:3000)                                  │
│   • API (localhost:3001)                                        │
│   • All processing happens here                                 │
└─────────────────────────────────────────────────────────────────┘
```

---

## Why This Architecture?

| Decision | Reason |
|----------|--------|
| Local processing | Customer data never leaves their machine |
| Cloud auth only | Single source of truth for licenses |
| Startup license check | No per-request latency |

---

## Component Responsibilities

| Component | Does | Does NOT |
|-----------|------|----------|
| Cloud API | Auth, billing, licenses | Process customer data |
| Local Dashboard | UI, processing, storage | Handle payments |
| Local API | Data ingestion, queries | Validate licenses |
````

````
---

## Security Model

```
User starts app
    │
    ├─► Verify license token (local, fast)
    ├─► Check expiry
    ├─► If valid → start services
    ├─► If expired → "Please re-authenticate"
    │
    ▼
Services run freely (no per-request checks)
```

---

## Changelog

| Date | Change |
|------|--------|
| 2026-02-11 | Added rate limiting documentation |
| 2026-02-10 | Updated architecture diagram |
| 2026-02-08 | Created INFRASTRUCTURE.md |
````

Key elements:

- "Always check this file" – tells Claude this is authoritative
- ASCII architecture diagram – visual overview
- "Why" tables – explains decisions, not just facts
- "Does / Does NOT" – prevents scope creep
- Changelog – tracks document evolution

The "Check Before Coding" Pattern

Put critical checks at the top:

```
# INFRASTRUCTURE.md

> **STOP. Before making ANY architectural changes:**
> 1. Read this entire document
> 2. Check if your change conflicts with existing decisions
> 3. If it does, discuss in this chat before proceeding

---
```

When to Create These Documents

DOCUMENT	CREATE WHEN	PURPOSE
`PROGRESS.md`	Project has phases	Track completion
`INFRASTRUCTURE.md`	Multiple services	Document architecture
`DECISIONS.md`	Complex tradeoffs	Record why, not just what
`CHANGELOG.md`	Long-running project	Track evolution

Linking Documents in CLAUDE.md

```
## Project Documentation

| Document | Purpose | When to Read |
|----------|---------|--------------|
| `INFRASTRUCTURE.md` | System architecture | Before architectural changes |
| `PROGRESS.md` | Current status | Before starting new work |
| `docs/SEPARATION.md` | Service boundaries | Before adding endpoints |

**ALWAYS read relevant docs before making cross-service changes.**
```

PART 26: WORKFLOW HACKS

These are tricks I've developed over 60+ projects that dramatically improve productivity.

1. Queue Mode for Testing Sessions

When you're testing and need to submit lots of small observations:

```
Claude, I'm going to be testing a lot of things now.
I want you to just take what I say and queue it.
When I'm ready for you to work again, I'll let you know.
Until then, whenever I submit a prompt (with or without screenshots),
just say "Ready for more".
Once I tell you I'm done, create a plan for everything in the queue.
```

This prevents Claude from trying to fix each issue individually, which wastes context.

2. Screenshots Are Your Best Friend

If you have a styling issue, don't dig through code hoping Claude understands your highlight. Take a screenshot.

Claude is amazing at reading screenshots. 99% of the time it just works and solves the issue.

Best practices for screenshots:

1. **Focus the area** – Don't screenshot your entire monitor. Crop to the specific element or section that's broken. Too much visual noise confuses Claude.

1. **Include the URL bar** – Whenever possible, make sure the browser's address bar is visible in the screenshot. This tells Claude exactly which page/route has the problem. Instead of saying "the button on the alerts page is broken," Claude can see `/dashboard/alerts` right in the image.

1. **One problem per screenshot** – If you have multiple issues, take multiple screenshots. Don't circle 5 things and expect Claude to track them all.

The WSL2 + VS Code problem:

If you run Claude Code in WSL2 through VS Code's terminal, CTRL+V doesn't paste images. Here's why:

- Your clipboard lives on Windows
- Claude Code runs in Linux (WSL2)

- VS Code's integrated terminal doesn't bridge image clipboard data between them
- Text pastes fine, but binary image data doesn't cross the boundary

My solution: I made clip2path – a PowerShell script that:

1. Grabs the image from Windows clipboard
2. Saves it to a WSL-accessible path
3. Copies the Linux path to your clipboard
4. You paste the path into Claude (not the image)

```
# Windows: CTRL+V clipboard image
# WSL Claude: paste → /mnt/c/Users/you/screenshots/clip-2026-02-11.png
```

Claude reads the file from the path. Problem solved.

3. Tell Claude "Don't Rush"

Claude loves showing you how fast it can get results. It doesn't care how it got there.

For complex features:

```
This is an important feature. Let's take it one step at a time.
If there is anything you're not 100% sure about, don't just execute — ask me fi
Don't rush. Quality over speed.
```

4. Explicit Deployment Gates

Claude loves to say "solved your prompt, let's deploy to production."

```
## ALWAYS
- ALWAYS ask before deploying to production
- NEVER auto-deploy, even if the fix seems simple
- NEVER assume approval — wait for explicit "yes, deploy"
```

PART 27: CLAUDE.MD RULES THAT SAVE YOUR ASS

Here's my battle-tested CLAUDE.md template. These rules have prevented countless issues.

The Numbered Critical Rules Pattern

Number your most important rules. Claude respects priority:

```
## Critical Rules

### 0. Test User - ALWAYS test@example.com

**The user account for ALL testing is `test@example.com`. NEVER assume your git

### 1. Missing UI? ALWAYS Check Feature Gates First

**When a UI element is missing — CHECK THE FEATURE MATRIX before assuming it's

- Feature gates control what each plan sees
- A trial = trial of the CHOSEN plan (Starter trial = Starter features)
- If the user's plan doesn't include the feature, the behavior is CORRECT

### 2. API Versioning - ALWAYS /api/v1/

**EVERY API endpoint MUST use `/api/v1/` prefix. NO EXCEPTIONS.**

```
CORRECT: /api/v1/users
WRONG: /api/users
```
```

The "Check X BEFORE Assuming Y" Pattern

Prevent Claude from jumping to conclusions:

```
### When Something Seems Wrong

- Missing UI element? → Check feature gates BEFORE assuming bug
- Empty data? → Check if services are running BEFORE assuming broken
- 404 error? → Check service separation BEFORE adding endpoint
- Auth failing? → Check which auth system BEFORE debugging
```

The "CORRECT vs WRONG" Format

Show examples of both. Claude learns patterns:

````
### Database Queries

```typescript
// CORRECT - aggregation pipeline
const result = await collection.aggregate([
  { $match: { userId } },
  { $limit: 1 },
]).toArray();

// WRONG - never use find()
const result = await collection.find({ userId }).toArray();
```

### API Paths

```
CORRECT: /api/v1/ai/ingest
WRONG:   /api/ai/ingest

CORRECT: POST to API service
WRONG:   POST to Dashboard service
```
````

The Service Separation Table

Quick reference prevents confusion:

```
## Service Separation

| Service | Purpose | URL | What Goes Here |
|---------|---------|-----|----------------|
| **Website** | Marketing | example.com | Landing, pricing, docs |
| **API** | Data ingestion | api.example.com | Ingest, webhooks |
| **Dashboard** | Customer UI | dashboard.example.com | Auth, billing, settings
| **Tasks** | Background jobs | (internal) | Cron, processors |

**READ `docs/SEPARATION.md` BEFORE making architectural changes.**
```

The Confusion Prevention Pattern

When multiple systems could be confused:

```
## Analytics Systems — Three Separate Things (NEVER Confuse)

| System | What It Tracks | Where Data Goes |
|--------|---------------|-----------------|
| **Visitor Analytics** | Dashboard users | Our DB |
| **Website Analytics** | Marketing visitors | Rybbit (external) |
| **Session Replay** | rrweb recordings | Our DB |

These are NOT the same. NEVER mix them up.
```

The Documentation Sync Checklist

Prevent drift:

```
## Documentation Sync

When updating any feature, keep these 4 locations in sync:

1. `README.md` (GitHub)
2. `packages/cli/README.md` (npm)
3. `packages/website/src/app/docs/` (website/docs)
4. `packages/cli/src/help.ts` (CLI help text)

If you update one, update ALL.
```

The Local Testing Requirements

Tell Claude what needs to run:

```
## Local Testing (MANDATORY)

**ALL services must be running:**

| Service | Port | Start Command |
|---------|------|---------------|
| Website | 3000 | `pnpm dev` |
| API | 3001 | `MONGODB_URI=... pnpm dev` |
| Dashboard | 3002 | `pnpm dev` |
| Tasks | — | `pnpm dev` |

### Common Mistakes

- Forgetting Tasks → violations never show up
- Forgetting API → no new data appears
- Wrong MONGODB_URI → services start but show empty
```

The Complete Template

````
# CLAUDE.md

## Critical Rules

### 0. [Your Most Important Rule]
[Details]

### 1. [Second Most Important]
[Details with CORRECT/WRONG examples]

### 2. [Third Most Important]
[Details]

---

## Service Separation

| Service | Purpose | Package |
|---------|---------|---------|
| ... | ... | ... |

**READ `docs/SEPARATION.md` BEFORE architectural changes.**

---

## Coding Standards

### [Language/Framework]

```typescript
// GOOD
[example]

// NEVER
[anti-pattern]
```

---

## [Specific System] — Multiple Things (NEVER Confuse)

| Thing | What | Where |
|-------|------|-------|
| A | ... | ... |
| B | ... | ... |
````

```
---

## Documentation Sync

When updating features, keep in sync:
1. [location]
2. [location]
3. [location]

---

## Local Development

| Service | Port | Command |
|---------|------|---------|
| ... | ... | ... |

### Common Mistakes
- [mistake] → [consequence]
- [mistake] → [consequence]
```

Why This Structure Works

1. **Numbered rules** – Priority is explicit
2. **Tables** – Quick reference, scannable
3. **CORRECT/WRONG** – Pattern matching, not memorization
4. **Common mistakes** – Anticipate failures before they happen
5. **Service separation** – Claude knows what goes where
6. **Documentation sync** – Prevents drift

This isn't just a config file. It's a **training document** for Claude. Every time Claude reads it, it learns your patterns.

QUICK REFERENCE

LESSON	SOLUTION
Renaming breaks things	Isolate until stable, use ENV vars
Plan mode contradictions	Name steps consistently, review for conflicts
Hooks don't wait	Keep fast, test independently, use exit code 2
Claude guesses types	Use TypeScript, always
Connection explosion	Database wrapper pattern
Shallow E2E tests	Explicit success criteria
Unstructured testing	Test MD template, CHECKLIST.md
Finding UI bugs	User-guided testing with ISSUE button
Claude ignores architecture	Authoritative docs with "STOP" pattern
Port conflicts	Fixed ports per service, test mode vs prod test mode
Multiple VS Codes	Kill ports before starting, dedicated port ranges
Re-explaining architecture	Commands as living documentation
Inconsistent scaffolding	Skills as templates with checklists
Tracking large projects	Progress reports with phase tables
Complex infrastructure	INFRASTRUCTURE.md with "check before coding"
Context loss	Reference MDs, not bloated CLAUDE.md
Claude rushes	"Don't rush, ask if unsure"
Unauthorized deploys	Explicit ALWAYS rules

TOOLS MENTIONED

TOOL	PURPOSE	LINK
clip2path	Screenshot to path for Claude	github.com/TheDecipherist/clip2path
claude-mv	Rename projects without losing context	curiouslychase.com
Vitest	Unit testing	vitest.dev
Playwright	E2E testing	playwright.dev

WHAT I SHIPPED USING THESE METHODS

These aren't theoretical tips. They come from shipping real projects:

- **Classpresso** – 59 stars, CSS utility tool
- **Wavesurf** – 600+ weekly downloads, audio player
- **RuleCatch** – AI governance platform (60+ test files, 500+ tests)

The patterns in this guide are what made those projects possible without losing my mind.

GITHUB REPO

All templates, hooks, skills, and CLAUDE.md examples:

github.com/TheDecipherist/claude-code-mastery

New in V5:

- Test MD templates
- ISSUES_FOUND.md / ISSUES_FIXED.md templates
- CHECKLIST.md template
- Architecture flowchart examples
- Authoritative document templates

SOURCES

Research & Issues Referenced

- Claude Code needs semantic code understanding – GitHub
- Plan mode not adhering – GitHub
- Plan mode ignores CLAUDE.md – GitHub
- Hooks not executing – GitHub
- Async hooks announcement – DevGenius
- TypeScript vs JavaScript for AI – Builder.io
- MongoDB connection pooling – MongoDB Docs
- Rescuing Claude conversations after rename – CuriouslyChase

Previous Guides

- V1: Global CLAUDE.md, MCP, Commands
- V2: Skills & Hooks
- V3: LSP Integration
- V4: MCP Tool Search, Custom Agents

What hard lessons have you learned? Drop your CLAUDE.md rules, test templates, and horror stories below.

Written by TheDecipherist – proving these methods work, one shipped project at a time.