Manual-Driven Development

190 Findings, 7 Hours, Zero Rule Violations

MARCH 10, 2026

THEDECIPHERIST.COM

TABLE OF CONTENTS

Every Claude Code session you have ever had started with Claude not knowing your system. It read a few files, inferred patterns, and started coding based on assumptions. At small scale that works fine. At production scale it produces confident, wrong code, and you do not find out until something breaks in a way that tests cannot catch, because Claude wrote the tests against its own assumptions too.

I call this confident divergence. It is the problem nobody in the AI tooling space is naming correctly. And it is the one that kills production codebases.

Manual-Driven Development fixes it. Here is what that looks like in production numbers:

Seven sections audited. 190 findings. 876 new tests written. 7 hours and 48 minutes of actual Claude Code session time against an estimated 234 to 361 hours of human developer time. That is a 30 to 46x compression ratio, reproduced independently across every section of a production codebase with 200+ routes, 80+ models, and a daemon enforcement pipeline that converts network policies into live nftables rules on the host.

And across all seven sections, not a single CLAUDE.md rule violated. Not one.

That last number is the one that should stop you. Everyone who has used Claude Code for more than a week has written CLAUDE.md rules and watched Claude ignore them three tasks later. The model does not do this deliberately. It runs out of context budget to honor them. MDD fixes the budget problem, and the rules hold. RuleCatch, which monitors rule enforcement in real time, reported 60% fewer rule violations during the SwarmK build compared to sessions running without MDD. Same model, same rules, same codebase. The only variable was MDD.

I am not going to ask you to take that on faith. The prompts that produced these results are published. The methodology is documented. The section-by-section data is in this article. Everything is reproducible.

If you are already using GSD or Mem0, you do not have to stop. MDD is a different layer solving a different problem. All three run together without conflict. I will explain exactly how near the end.

THE PROBLEM NOBODY IS NAMING CORRECTLY

When Claude Code produces wrong code at scale, the community tends to blame one of two things: context rot, where quality degrades as the session fills up, or session amnesia, where Claude forgets everything when the session ends. GSD was built to solve context rot. Mem0 and Claude-Mem were built to solve session amnesia. Both are real problems. Both tools are real solutions.

But there is a third problem that neither tool addresses, and it is the one that produces confident divergence.

Claude does not know your system. Not in the way you do. It reads a few files, infers patterns, and starts coding based on assumptions. At production scale, with 200+ routes, 50+ models, and business rules distributed across a codebase that took months to build, the inferences diverge from reality. Claude produces code that compiles, passes its own tests, and is confidently wrong.

Here is what makes confident divergence so hard to catch: everything looks correct. The code runs. The tests pass. Claude wrote the tests against its own assumptions about what the system does, not against what the system actually does. The divergence only surfaces in production, when a real user hits the edge case Claude never knew existed.

Here is what makes it so hard to prevent: the problem is not just that Claude does not know your system. It is that you cannot reliably narrate your system to Claude either.

You built the whole thing. You know how operator scoping works, how the tier hierarchy enforces access, how tunnels allocate /30 subnets in the 10.99.x.0 range. You know all of it in theory. But when you sit down to write a prompt at 11pm, you will not remember to mention that operators are scoped to specific groups and cannot modify policies outside their assigned groups. You will forget that ROLE_HIERARCHY is defined in three different files. You will not think to tell Claude that base-tier policies are system-only and cannot be created via the API.

You are not going to enumerate 200 routes worth of business rules in a prompt. Nobody can.

So Claude guesses. And confident divergence happens.

That is the problem MDD solves. Not context rot within a session. Not forgetting between sessions. The deeper problem of Claude not having explicit knowledge of your system in the first place.

THE TOKEN OBSESSION IS SOLVING THE WRONG PROBLEM

Before explaining MDD, it is worth naming something about the current tooling landscape, because the framing most tools use will make MDD seem like another entry in the same race. It is not.

Every tool launched in the last twelve months leads with the same promise: fewer tokens, lower cost, faster responses. Mem0 claims 90% token reduction. Zep claims 90% latency reduction. GSD keeps your main context at 30-40% by offloading work to fresh subagents. The implicit argument is always the same: the bottleneck is tokens, so the solution is to use fewer of them.

This framing is wrong. Not because tokens do not matter, but because it misidentifies the bottleneck.

MDD saves tokens. When Claude has an explicit documentation file describing exactly how a feature works, it does not need to read fifteen source files to reconstruct the same picture. You use fewer tokens naturally. But that is the exhaust, not the engine. The engine is accuracy. Token efficiency is what happens when Claude stops guessing.

If you believe the bottleneck is tokens, you build token compression tools. If you believe the bottleneck is knowledge, that Claude fails not because it runs out of context but because it never had accurate information about your system in the first place, you build documentation infrastructure. These are fundamentally different bets.

On the published numbers: The 90% token reduction figure that Mem0 publishes is real but carefully framed. The comparison baseline is stuffing a full 26,000-token conversation history into every request, which is the most wasteful possible approach. Against that baseline, almost any selective retrieval system looks miraculous. The benchmark was designed and run by Mem0's own team. Competitors Letta and Zep have both publicly challenged the methodology. Zep's reanalysis found configuration

discrepancies that inflated the scores. And Mem0's own research paper buries a real tradeoff: at 30 to 150 session turns, it accepts a 30 to 45 percentage point accuracy drop on implicit and preference tasks. Token savings at the cost of accuracy is a legitimate engineering tradeoff. It is not the same as being more accurate, which is how the tool is marketed.

GSD makes no explicit token claim and does not try to. Its argument is architectural and plausible. Fresh subagent contexts prevent context rot. But there is no external benchmark or controlled study proving the quality improvement. The evidence is anecdotal, the adoption is real, the mechanism is sound. Plausible and popular is not the same as measured.

None of this is an argument against either tool. It is an argument for being clear about what problem you are actually solving, because the problem MDD solves is different from the problem both of them solve.

WHAT MDD ACTUALLY IS

MDD stands for Manual-Driven Development. It is a convention set, not a framework. No installer, no config file, no CLI to learn. Three things:

1. A documentation handbook, one markdown file per feature, written before code
2. A CLAUDE.md lookup table that maps feature areas to their documentation files
3. A phased workflow: Audit, Document, Implement, Test, Verify, Ship

The core insight is that documentation is context compression.

Without docs, Claude reads 10 to 15 source files, roughly 15,000 to 20,000 tokens, to piece together how a feature works, and still misses the connections between them. With a focused markdown doc, Claude reads one file, roughly 2,000 to 3,000 tokens, and has the complete picture. That savings compounds across every task.

The stack:

LAYER	PURPOSE
CLAUDE.md	Rules, hooks, banned patterns
Hooks	Deterministic enforcement
Documentation Handbook	One markdown per feature
YAML Frontmatter	Scannable dependency graph
Lookup Table	CLAUDE.md maps features to docs
Review Prompts	Verification sweeps

The phased workflow:

Audit first. Before writing anything, have Claude crawl the existing codebase and document what actually exists. Do not assume you know your own app. The SwarmK audit found roughly 15% of features were broken or half-implemented. No documentation would have helped if it described code that did not work.

Document before code. For each feature, Claude writes a spec first. One file per feature. The doc defines data models, endpoints, business rules, edge cases, edition gating, and cross-references. The doc is the only deliverable of this step. No code changes.

Implement from the doc. Claude reads the doc it just wrote, then codes to match the spec. If implementation reveals the spec was wrong, update the doc first.

Test the doc's claims. If the doc says DELETE returns 409 when dependencies exist, there must be a test for exactly that.

Verify. Claude reads each doc against actual source code and fixes discrepancies. Code is truth. Docs match code.

Ship everything together. Doc plus code plus tests in the same git commit.

WHAT ACTUALLY CHANGES IN EVERY SESSION

The compression ratio, 30 to 46x, is the headline number. But the more important thing MDD produces is not faster audits. It is Claude that starts tasks instantly, makes fewer mistakes, and actually follows the rules you wrote. In every session. Consistently.

These three outcomes are connected and they all come from the same root cause: Claude arrives at actual work with most of its context available instead of a fraction of it.

Tasks start faster. Before MDD, starting any non-trivial task meant Claude spending the first portion of its context budget doing archaeology. Opening files, tracing imports, piecing together what depends on what, reconstructing business rules from implementation details. That exploration phase is expensive and lossy. Claude frequently got it partially wrong even after reading everything, because the relationships between components were implicit.

With MDD documentation in place, that phase disappears. Claude reads one file and has the complete picture: data models, endpoints, business rules, dependencies, edition gating, cross-references, known edge cases. It does not need to infer that operators are scoped to specific groups and cannot modify policies outside their assignments. It reads that statement directly. Task startup goes from minutes of exploration to immediate execution.

Fewer mistakes because Claude knows what depends on what. The most damaging Claude Code errors are not syntax errors or logic errors, those are visible. The damaging errors are the ones where Claude implements something correctly in isolation but breaks something it did not know was connected. It changes a model field, does not realize three other features read that field with specific assumptions, and introduces a silent data integrity issue that passes all tests. Confident divergence at the implementation level.

MDD documentation includes explicit dependency graphs in YAML frontmatter. Every feature doc declares what it depends on and what depends on it. When Claude has that graph loaded before it writes a single line, it cannot unknowingly break a dependency. The connection is explicit, not inferred.

Claude follows CLAUDE.md rules because it has context left to do so. This is the result that matters most and gets talked about least.

CLAUDE.md rules are not magic. Claude reads them at the start of a session and then works within a shrinking context window. As that window fills with file reads, tool calls, conversation history, and code output, the rules compete for attention with everything else Claude is tracking. In a bloated session, Claude does not deliberately ignore your rules. It runs out of room to honor them.

Since adopting MDD: zero CLAUDE.md violations across seven production audit sections. Not one. RuleCatch tracked this in real time and recorded 60% fewer violations compared to sessions running without MDD. Same model. Same rules. Same codebase. The only variable was MDD giving Claude enough context budget to actually follow what you told it to do.

This is where the two tools connect naturally. MDD gives Claude the context budget to follow your rules. RuleCatch provides real-time enforcement for when a rule is at risk of slipping anyway. MDD is structural. RuleCatch is the safety net. Together they close the loop between "I defined a rule" and "that rule was actually followed."

THE `.MDD/.STARTUP.MD` FILE: TWO ZONES, ONE FILE

There is an important distinction between what MDD needs from session continuity and what memory tools provide. The best way to see it is through one file.

Mem0 and Claude-Mem capture what happened: session history, tool observations, coding preferences learned over time. That is episodic memory and it is genuinely useful. But `.startup.md` captures something different. What is currently true about this system, and what are the standing decisions Claude needs to know before touching anything.

"Do not modify the nginx upstream block until E2E tests pass" is not a memory of a conversation. It is an operational constraint. A memory tool cannot capture it because it was never said in a session. It was decided, and decisions live in your head until you write them down somewhere Claude will actually read them.

`.startup.md` is where you write them down.

The file has two zones separated by a single divider line. Everything above the divider is auto-generated. Everything below it is yours and automation never touches it.

The auto-generated zone is rebuilt automatically by MDD after every status check, every audit, and every fix cycle. It always reflects current project state:

```
Generated: 2026-03-10
Branch: feat/webserver-ssl
Stack: Node.js / TypeScript / MongoDB / Docker Swarm

Features Documented: 52 files
Last Audit: 2026-03-08 (190 findings, 187 fixed, 3 open)

Rules Summary:
  - No direct req.body spread into $set
  - All endpoints require company_id scoping
  - Commit gate: doc + code + tests in same commit
```

Claude reads this and instantly knows where the project stands. No archaeology. No file navigation. The session starts with accurate project state already loaded.

The Notes zone is append-only. When you run `/mdd note "do not touch the nginx upstream block until E2E tests pass"`, MDD appends a timestamped entry below the divider. The next session starts with Claude reading both zones, machine-generated state and your human decisions together.

```
- [2026-03-08] tenant isolation fix verified in production, safe to proceed
- [2026-03-09] Playwright E2E suite planned for all SSL config combinations
- [2026-03-10] do not modify nginx upstream block until E2E tests pass
```

Three subcommands manage it:

- `/mdd note "text"` appends a timestamped entry
- `/mdd note list` prints only the Notes section
- `/mdd note clear` wipes the Notes section after confirmation

Notes are the one thing in the MDD system that Claude will not regenerate if you delete them. They exist only because you wrote them.

THE FAILURE THAT INVENTED THE TWO-PROMPT

ARCHITECTURE

The most important technical innovation in MDD was not designed. It was discovered by watching a session die.

SwarmK's networking stack covers 29 distinct feature areas: policies, groups, traffic flows, encryption tunnels, rate limiting, bandwidth, load balancing, proxy layer, DNS, WAF, SSL, CSP scanning, location profiles, Docker networks, topology, connections. The original audit prompt tried to handle all of it in one shot. Four phases. 100+ files. One prompt.

It lasted fifteen minutes.

Claude worked through Phase 1 (planning) and started Phase 2 (source code). By the time it reached the daemon files, the context window was full. It compacted. The compaction summary preserved the general intent of what it had read but destroyed the specifics. Exact field names, precise validation logic, the nuances of how business rules were actually implemented versus how they were supposed to be implemented. Claude compacted a second time. By Phase 4 (report writing), it was working from summaries of summaries. Fifteen minutes of session time. Nothing usable. Not a single finding written down.

That is confident divergence at the tooling level. The session looked like it was working until the moment it produced nothing.

The realization that came from watching it fail: context compaction destroys specifics but cannot touch the filesystem. Anything written to disk before compaction happens is completely safe. The problem with the single prompt was that Claude was accumulating everything in memory, planning to write it all at once at the end. When compaction hit, the accumulated work was gone.

The fix was simple in retrospect. Split the work. One prompt that does nothing except read source files and write notes to disk after every single feature, before moving to the next one. A second prompt that reads only the notes file and produces the report.

The critical instruction in Prompt 1:

> *"After processing EACH feature, immediately append your notes to the file. Do NOT hold findings in memory waiting to write them all at once. If context compacts, everything not yet written to file is LOST."*

Prompt 2 reads only the notes file. Not the source files. The notes file compressed 100+ source files into roughly 8,000 tokens. Prompt 2 has 192,000 tokens available for analysis and produces the full findings report in 4 minutes.

	SINGLE PROMPT (FAILED)	TWO-PROMPT MDD
Compactions	2 (died in Phase 2)	1 (survived, 1,159 lines safe on disk)
Output	Nothing	1,626 lines of notes + 363-line report
Time	~15 min before killed	24 min total
Findings	None	6 CRITICAL, 4 HIGH, 9 MEDIUM, 6 LOW

We ran this architecture across 7 sections of SwarmK. It survived 3 to 4 compactions per run with zero data loss every time. The methodology works because it manages context mechanically, by making disk the default state instead of memory. If it worked on networking (33 features, 100+ files) it works on any section.

THE NETWORKING AUDIT: THREE REAL PROMPTS

Prompt 1: Read and Notes

```
You are running Phase 1 of an MDD audit on the [SECTION] section.

Read each source file in order. After processing EACH feature, immediately
append structured notes to plans/[section]-raw-notes.md. Do NOT hold
findings in memory waiting to write them all at once. If context compacts,
everything not yet written to file is LOST.

For each feature, note:
- Endpoints (method, path, auth requirements)
- Data model fields and whether company_id scoping exists
- Business rules enforced in code (specific, cite actual checks)
- Agent/daemon handlers or "API-only, no daemon enforcement"
- Test coverage (count and what they actually cover)
- Red flags (missing validation, scope bypass risks, error handling gaps)

After processing EACH feature, append immediately. Do not wait.
```

Prompt 2: Analyze and Report

```
Read plans/[section]-raw-notes.md in full.
Do NOT read source files. Everything you need is in the notes.

Produce a structured findings report at plans/[section]-findings.md with:

1. Executive summary
2. Feature completeness matrix
3. Findings sorted by severity (CRITICAL to LOW)
4. For each finding: description, affected files, business impact,
   fix recommendation, fix complexity estimate
5. Pipeline analysis (for sections with enforcement pipelines)
6. Test coverage gaps
7. Recommended fix order (P0/P1/P2/P3)

CRITICAL = security vulnerability, data integrity risk, or production breakage
HIGH = incorrect behavior, missing enforcement, or significant test gap
MEDIUM = quality issue, validation gap, or performance concern
LOW = cleanup, documentation gap, or minor inconsistency

Output the report. Do not start writing fixes.
```

Prompt 3: P0 Security Fixes

The fix prompt does not ask Claude to figure anything out. It tells Claude exactly what is broken (read the audit findings), what should exist (read the feature docs), and how it is done correctly elsewhere (read policies.ts, which already has the correct pattern, and apply it to the affected routes).

The 7 specific fixes from the networking audit:

- ratelimit-service.ts: no company_id in query, no requireMinRole
- bandwidth-service.ts: same problem
- lb-service.ts: same problem
- connections.ts: no company_id in the $match stage of the aggregation pipeline
- policy-history-recorder.ts: accepts company_id as a parameter but never writes it to the document
- Parent routes (ratelimit.ts, bandwidth.ts, lb.ts): verify authenticate plus requireMinRole exist
- All three service PUT endpoints: spreading req.body into $set (mass assignment vulnerability)

Every fix lists the specific file, the specific issue, and the specific fix. Every fix gets three tests: tenant isolation (Company A user cannot see Company B data), RBAC (Viewer cannot PUT or DELETE, Operator can), and mass assignment (sending _id or company_id in the PUT body does not change those fields). Docs ship in the same commit as the code.

Output: 6 CRITICAL issues resolved, 52 new tests, 65/65 passing, TypeScript clean, 4 docs updated. 13 minutes.

THE COMPRESSION RATIO PROOF: SEVEN SECTIONS, FULL DATA

SECTION	FINDINGS	ESTIMATE	ACTUAL	COMPRESSION
Networking	25	37-52 hr	65 min	34-48x
Servers	25	32-54 hr	81 min	24-40x
Projects	27	19-34 hr	71 min	16-29x
WebServers	**39**	**45-74 hr**	**58 min**	**47-77x**
Agents	33	47-72 hr	53 min	53-82x
Providers	20	29-35 hr	55 min	32-38x
Volumes	21	25-40 hr	85 min	18-28x
Total	**190**	**234-361 hr**	**468 min (7h 48m)**	**30-46x**

The WebServers row is the one worth staring at. 39 findings, the most of any section, completed in 58 minutes, less time than any other section despite having the most findings. That is what happens when Claude has a complete map of the system before it starts. It does not slow down as complexity increases.

Combined output across all seven pipelines:

- 876+ new tests written
- 3,945 total tests passing (up from roughly 3,200 before audits)
- servers.ts split from 1,169 lines to 576 across 5 focused files
- Tenant isolation fixed across 4 routes plus a full WebSocket handler rewrite
- volume.prune scoped to managed resources only (it was silently deleting ALL Docker volumes)
- LVM shell injection blocked
- Backup directory path traversal prevented
- Versioned encryption key rotation with backward-compatible migration
- Privilege escalation guard on auth provider auto-provisioning

The compression comes from eliminating wasted tokens. Human developer time estimates assume reading unfamiliar code, investigating bugs without a complete picture, writing tests against assumed behavior, and debugging when implementation diverges from intent. MDD eliminates all four. Claude does not investigate, assume, or debug. It reads and applies. No confident divergence.

TEN LESSONS FROM REAL FAILURES

These are not principles. They are postmortems. Every one came from a real session doing the wrong thing.

Lesson 1: Agents skip documentation. A prompt said "fix issues AND write documentation simultaneously." Claude wrote all the code fixes, wrote zero documentation files, and said done. Never give Claude a prompt where documentation is a side task alongside code.

Lesson 2: Parallel agents produce plausible but wrong docs. 8 parallel agents wrote 52 docs. Verification found 6 discrepancies including 5 wrong edition classifications. Each agent worked from partial context and produced plausible-sounding but incorrect content. Verification must be single-threaded.

Lesson 3: Edition gating defaults to "Both." Writing agents defaulted features to "Both" (OSS + Cloud) when 5 were actually Cloud-only. They did not check app.ts. Edition must be verified from route mounting, never from assumptions.

Lesson 4: Claude tries to commit to main. During doc verification, Claude tried to commit directly to main. The check-branch.sh hook blocked it. Hooks are guarantees. CLAUDE.md rules can be ignored under context pressure. Hooks cannot.

Lesson 5: Context compression beats code navigation. Same task with and without a doc: 15,000 tokens versus 2,000 tokens, and the doc version produced correct code while the navigation version did not. Always read the doc first.

Lesson 6: Agents are safe for extraction, not verification.

TASK TYPE	AGENTS SAFE?	WHY
Writing docs from source code	NO	Must cross-reference multiple files
Verifying docs against code	NO	Must trace business rules across files
Adding frontmatter to verified docs	YES	Extraction, not judgment
Formatting, linting, template application	YES	Mechanical transformation
Code fixes from a fix plan	MAYBE	Safe if fixes are independent

If the task requires judgment about whether something is correct, do not parallelize it.

Lesson 7: "Done" is self-assessed and unreliable. Claude said the phase was done. It had written code fixes but zero documentation files. Add file-existence checks as commit gates.

Lesson 8: Explicit reference data beats instructions. Telling an agent "check app.ts for requireEdition()" is an instruction it might deprioritize under context pressure. Giving it a list of 21 specific features that must be "cloud" is reference data it can verify against mechanically. A lookup list beats a procedure.

Lesson 9: Massive audits need a read prompt and a write prompt. The original single-prompt audit died twice. The two-prompt version produced 1,626 lines of notes plus a 363-line report in 24 minutes. More than 30 source files means two prompts.

Lesson 10: The full pipeline works. Audit to fix in 37 minutes. 6 CRITICAL tenant isolation vulnerabilities resolved. Audit estimated 6 to 8 hours. Actual: 13 minutes. Write fix prompts that reference both the audit findings and a working reference implementation.

WHERE MDD FITS ALONGSIDE OTHER TOOLS

Three problems. Three tools. None of them the same.

GSD solves context rot, the quality degradation that happens as a session fills up. It routes around the problem by spawning fresh subagent contexts for each task, keeping your main orchestrator lean while subagents do the heavy lifting in clean 200K-token windows. Strong on greenfield, autonomous execution, and forward momentum on new features.

Mem0 / Claude-Mem solve session amnesia, Claude starting every session with zero knowledge of who you are or what you built. Memory tools capture session history, preferences, and observations, then inject relevant context into future sessions. Strong on preference persistence and eliminating the exploration phase across multi-day work.

MDD solves confident divergence, Claude not knowing your system well enough to be trusted with it. Documentation infrastructure that makes the right knowledge explicit, available, and impossible for Claude to misinterpret. Strong on brownfield audits, production codebases, and any situation where Claude getting the wrong answer is worse than Claude going slowly.

All three can run together. MDD runs continuously as your documentation foundation. Memory tools run in the background. GSD runs for discrete new feature phases. The only practical consideration: at session start, MDD docs, memory injection, and GSD planning state may all compete for context budget. Prioritize MDD docs, they are the most precise, and tune memory injection downward if sessions start heavy.

The recommended sequence for a new project: run MDD first, build the documentation handbook, fix CRITICAL findings. Add a memory tool so it starts building session history from a clean baseline. Add GSD when you begin a significant new feature phase and point it at your existing MDD docs.

The one-sentence summary of each:

- **GSD:** Solves the problem of Claude getting worse as a session gets longer.
- **Mem0 / Claude-Mem:** Solves the problem of Claude forgetting everything between sessions.

- **MDD:** Solves the problem of Claude not knowing your system well enough to be trusted with it.

All three problems are real. Most developers are treating them as one problem and getting frustrated when a single tool does not fix all three.

THE PROMPT LIBRARY

These are the actual prompts used on SwarmK. Adapt file paths to your project.

Audit P1: Read and Notes

```
You are running Phase 1 of an MDD audit on the [SECTION] section.

Read each source file in order. After processing EACH feature, immediately
append structured notes to plans/[section]-raw-notes.md. Do NOT hold
findings in memory waiting to write them all at once. If context compacts,
everything not yet written to file is LOST.

For each feature, note:
- Endpoints (method, path, auth requirements)
- Data model fields and whether company_id scoping exists
- Business rules enforced in code (specific, cite actual checks)
- Agent/daemon handlers or "API-only, no daemon enforcement"
- Test coverage (count and what they actually cover)
- Red flags (missing validation, scope bypass risks, error handling gaps)

After processing EACH feature, append immediately. Do not wait.
```

Audit P2: Analyze and Report

```
Read plans/[section]-raw-notes.md in full.
Do NOT read source files. Everything you need is in the notes.

Produce a structured findings report at plans/[section]-findings.md with:

1. Executive summary
2. Feature completeness matrix
3. Findings sorted by severity (CRITICAL to LOW)
4. For each finding: description, affected files, business impact,
   fix recommendation, fix complexity estimate
5. Pipeline analysis (for sections with enforcement pipelines)
6. Test coverage gaps
7. Recommended fix order (P0/P1/P2/P3)

CRITICAL = security vulnerability, data integrity risk, or production breakage
HIGH = incorrect behavior, missing enforcement, or significant test gap
MEDIUM = quality issue, validation gap, or performance concern
LOW = cleanup, documentation gap, or minor inconsistency

Output the report. Do not start writing fixes.
```

P0 Fix Prompt Template

```
Read plans/[section]-findings.md.
Read documentation/[project]/[relevant-feature].md.
Read src/server/routes/[reference-implementation].ts. This file already
has the correct pattern. Apply the same pattern to the affected routes.

Fix all CRITICAL findings:
[paste CRITICAL findings from the report here]

Requirements:
- Create feature branch: fix/[section]-critical
- Write tests for every fix (tenant isolation, RBAC, mass assignment)
- Update affected documentation files
- TypeScript must compile clean
- All existing tests must still pass
- Commit: "fix([section]): resolve CRITICAL findings from audit"

When done: run full test suite, report pass count.
```

Documentation Verification Prompt

```
Review documentation/[project]/[feature-doc].md against actual source code.

Read the doc, then read every source file in its frontmatter owner section.

Check:
1. Every endpoint exists with correct method, path, and auth
2. Every data model field is present with correct type and constraints
3. Business rules in the doc match actual implementation
4. Edition gating matches app.ts route mounting, not just the doc's assertion
5. Cross-references to other docs are still accurate

Fix discrepancies. Code is truth. Update doc to match reality.
Update status to "verified" and last_verified date.
```

QUICK REFERENCE

MDD file structure

```
project/
  .mdd/                          # Machine state (gitignored)
    .startup.md                  # Two-zone session context file
    docs/                        # Feature documentation
      00-architecture.md         # System overview
      01-[feature].md            # One file per feature
    audits/                      # Audit working files
      notes-[date].md            # P1 output
      report-[date].md           # P2 output
  CLAUDE.md                      # Includes lookup table
```

CLAUDE.md additions for MDD

```
## MDD Documentation Handbook

Before working on ANY feature, read the relevant doc:

| Feature | Doc |
|---------|-----|
| [Feature] | .mdd/docs/[NN]-[feature].md |

## MDD Rules

- NEVER write code without reading the feature doc first
- If no doc exists for a feature you are modifying: write the doc first
- Audit notes: append after EACH feature, never hold in memory
- Fix prompts: always include audit findings + feature doc + reference implemer
- Ships: doc + code + tests in the same commit, always
```

YAML frontmatter schema

```
---
id: "12-policies"
title: "Network Policies"
edition: "cloud"
status: "verified"
last_verified: "2026-03-10"

owner:
  routes:
    - "src/server/routes/policies.ts"
  models:
    - "src/core/models/policy.ts"

depends_on:
  - id: "02-authentication"
    reason: "All endpoints require JWT auth"

used_by:
  - id: "48-daemon"
    reason: "Daemon generates nftables rules from policies"

collections:
  - "policies"

endpoints:
  - "GET /api/v1/policies"
  - "POST /api/v1/policies"
  - "DELETE /api/v1/policies/:policyId"
---
```

Claude can scan frontmatter across all 52 docs in roughly 500 tokens total, the entire dependency graph without loading any full doc.

TheDecipherist, March 2026.