Manual-First Development

A deep-dive into the workflow that turns Claude from a code generator into a reliable engineering partner

APRIL 20, 2026

THEDECIPHERIST.COM

TABLE OF CONTENTS

> *A deep-dive into the workflow that turns Claude from a code generator into a reliable engineering partner.*

> *## The Real-World Proof*
> *> Since I started using MDD across all my projects, I have not hit Claude's usage limits once, not even close.*
> *> A typical session: 13% current usage, 19% weekly. Running multiple VS Code windows with multiple Claude Code terminals simultaneously, every single day.*
> *> Before MDD, I was burning through context constantly, Claude losing track, rewriting things it just wrote, hallucinating integrations that didn't exist. Every session started over from scratch. The usage would spike because Claude was spinning its wheels redoing work it had already done.*
> *> After MDD, the sessions are tighter. Claude reads the doc, knows exactly what was built and why, and gets straight to work. The output quality is night and day. The token efficiency follows naturally.*
> *> This isn't a coincidence. When Claude has structured project memory to work from, it stops guessing and starts engineering.*

THE PROBLEM WITH "JUST USING CLAUDE"

Here's a scenario that probably sounds familiar. You describe a feature, Claude writes code, the code compiles, the tests pass, and everything feels great. Then six weeks later something breaks in a way that makes no sense. Or Claude confidently rewrites a function that was quietly holding three other features together. Or you ask about a piece of functionality you built together last month, and Claude has no idea what you're talking about because that context vanished the moment you closed the tab.

This isn't a fluke. It's the nature of how Claude works. Claude is a context-window-scoped collaborator. It knows what you told it today. It has no memory of what you decided last week, no understanding of how features depend on each other, no stake in

what happens after the conversation ends. When you close the tab, all that accumulated context, the tradeoffs you discussed, the edge cases you accounted for, the architecture decisions you made, just disappears.

MDD (Manual-First Development) is built specifically to solve this problem. It's a structured workflow that trades the illusion of speed for actual reliability. The core idea is that documentation isn't overhead, it's the mechanism by which Claude accumulates project memory that survives across sessions. It's also how you, the human, stay in control of what actually gets built.

WHAT MDD IS (AND ISN'T)

MDD is not a documentation framework. It's not a process for writing specs before code. It's not a bureaucratic gate designed to slow you down.

It's a **shared working model** between you and Claude. Every feature you build creates a structured document capturing intent, data flow, contracts, and known edge cases. That document becomes Claude's memory. When you come back three weeks later, Claude doesn't start from scratch, it reads the doc and picks up exactly where you left off. When something breaks, the doc tells you what it was supposed to do. When a new feature needs to integrate with an old one, the doc tells Claude exactly what contracts it's working with.

The workflow ships as the `/mdd` command inside the Claude Code Mastery Project Starter Kit, but the concepts apply anywhere you're using Claude for serious development.

THE CORE LOOP: DOCUMENT, AUDIT, FIX, VERIFY

Every MDD interaction follows four phases. Once you understand this loop, everything else in the workflow makes sense.

Document first. Before any code is written, MDD creates a structured document describing the feature: its purpose, data model, API contracts, business rules, known edge cases, and which other features it depends on. This isn't a draft that gets thrown

away, it becomes the source of truth that all subsequent work gets measured against.

Audit the result. After implementation, MDD audits the code against the documentation. Both you and Claude have a clear picture of what "correct" means. Discrepancies between what the doc says and what the code does are findings, not debates.

Fix against the doc. When fixes are needed, Claude works against the documented spec. Tests are never modified to make code pass, code is modified to satisfy the spec. If the spec seems wrong, the doc gets updated first, then the code follows. The order matters.

Verify against reality. MDD doesn't consider a feature done when tests pass. It requires integration verification against the real runtime environment: actual HTTP calls, real database writes, actual browser behavior. "Tests pass" is not the finish line.

THE `.MDD/` DIRECTORY: WHERE PROJECT MEMORY LIVES

The first time you run `/mdd` in a project, MDD creates a `.mdd/` directory with three subdirectories:

```
.mdd/
├── docs/          # Feature documentation files (the source of truth)
├── audits/        # Audit reports, drift scans, flow analyses
└── ops/           # Deployment runbooks
```

It also creates `.mdd/.startup.md`, an automatically maintained project snapshot. Every time you run any MDD command, this file gets rebuilt with fresh data: which features are documented and at what status, when the last audit ran and what it found, which branches are active. This is what gets injected into every new Claude session, so Claude immediately understands the state of the project without you having to explain it.

The `audits/` directory is git-ignored by default. MDD adds it to `.gitignore` automatically, because audit reports are ephemeral and regenerated on demand. You don't want dozens of timestamped audit files cluttering your git history.

BUILD MODE: BUILDING FEATURES THE RIGHT WAY

The default mode, activated by running `/mdd`, is Build Mode. This is where you'll spend most of your time.

Worktree Isolation (Step 0)

The first thing MDD does is offer to create a worktree. A worktree is a separate checkout of the repository in a sibling directory with its own branch. This lets you run multiple `/mdd` sessions in parallel without them stepping on each other. Session A is building the auth system while Session B is building the payment flow, and neither one touches the other's files or bleeds into the other's Claude context.

This matters more than it sounds. When you're deep in a feature that touches a lot of files, you really don't want another session's edits showing up in your working directory mid-build. Worktrees make that impossible.

Context Gathering with Parallel Agents (Phase 1)

Before asking you a single question, MDD launches three parallel Explore agents at the same time:

- **Agent A** reads `CLAUDE.md` and your project architecture docs, pulling out coding rules, quality gates, port assignments, and conventions
- **Agent B** reads all existing `.mdd/docs/*.md` files and builds a map of existing features, their statuses, and their dependency chains
- **Agent C** scans `src/` to understand the directory structure, key files, and the detected tech stack

They all run at once and return simultaneously. MDD synthesizes their output before asking you anything. So when it asks "does this feature depend on any existing features?", it already knows what your existing features are. When it asks about API

endpoints, it already knows your versioning conventions. These aren't generic questions getting copy-pasted into a chat, they're informed questions backed by context that was just gathered in parallel.

MDD also detects whether the feature is a **tooling task** (a command, hook, config change, or workflow) versus an application feature. Tooling tasks skip the database and API questions entirely since they're irrelevant, which keeps things moving.

Data Flow Analysis (Phase 2)

This is the phase that separates MDD from anything else in the AI-assisted development space. It's also the one that prevents the most bugs.

Before writing any documentation, MDD traces exactly what data this feature will consume, transform, and display. For each piece of data it follows the full chain:

1. **Backend origin**, where is this value computed? What formula? Which file and line?
2. **API transport**, what's the exact shape in the response? Is it typed correctly?
3. **Frontend consumption**, how does the UI receive and use this value? Is there any transformation between what the API returns and what gets shown?
4. **Parallel computations**, is this same concept computed anywhere else in the codebase? Does it use the same logic?

Then MDD runs an impact analysis. For every endpoint or function the feature will modify, it greps for all existing usages. Every other file that consumes what you're about to change gets listed. Those are the files that might break silently once your change goes in.

The findings land in front of you as a gate. You see exactly what MDD found before a single line of documentation is written. If there are consistency issues, places where the same concept is computed two different ways in two different parts of the codebase, you deal with them now instead of six weeks from now after you've built on top of them.

Greenfield projects skip this phase entirely. If there are no existing docs and fewer than five source files, there's nothing to trace yet.

The Feature Document (Phase 3)

With context gathered and data flow understood, MDD writes the feature document. The structure is strict:

```
---
id: <NN>-<feature-name>
title: <Feature Title>
edition: <project name>
depends_on: [list of feature doc IDs]
source_files:
  - src/handlers/orders.ts
  - src/types/orders.ts
routes:
  - POST /api/v1/orders
  - GET /api/v1/orders/:id
models:
  - orders
test_files:
  - tests/unit/orders.test.ts
data_flow: .mdd/audits/flow-orders-2026-04-20.md
last_synced: 2026-04-20
status: draft
phase: documentation
mdd_version: 4
known_issues: []
---
```

Every field in this frontmatter is doing something specific. `source_files` tells Scan Mode where to look for drift. `depends_on` tells Graph Mode what the dependency chain looks like. `last_synced` tells Scan Mode when to start checking for commits. `mdd_version` tracks which version of the MDD workflow was used, which enables upgrade paths when the workflow itself changes over time.

The document body covers purpose (why does this exist?), architecture (how does it fit the system?), data model (exactly what's stored?), API contracts (what does each endpoint accept and return?), business rules (what validation, what state machines, what invariants?), data flow, dependencies, and known issues.

MDD shows you the completed doc and asks if it accurately describes what you want to build. You can revise it before any code is written. This is the most important gate in the entire workflow.

Test Skeletons (Phase 4)

From the documented endpoints, business rules, and edge cases, MDD generates test skeletons. These are placeholder tests that describe the expected behavior but contain no implementation at all.

```
describe('Orders', () => {
  describe('createOrder', () => {
    it('should create order and return 201 with order ID', async () => {
      // Arrange
      // Act
      // Assert — minimum 3 assertions based on documented response shape
      expect.fail('Not implemented — MDD skeleton');
    });

    it('should return 400 when items array is empty', async () => {
      expect.fail('Not implemented — MDD skeleton');
    });
  });
});
```

When both unit tests and E2E tests are needed, MDD launches two parallel agents to write them at the same time. They go in different files, so there's no write conflict.

The Red Gate (Phase 4b)

Right after generating skeletons, MDD runs the new tests. Every single one must fail. This is not optional and has no skip condition.

The Red Gate exists because "green before you start" is meaningless. If a skeleton passes before you've written any code, either the test has no real assertion or existing code already satisfies it, which means you're testing the wrong thing. MDD investigates every unexpected pass and requires it to be resolved before implementation can begin.

```
🔴 Red Gate: 8/8 failing (expected)
   All skeletons confirmed RED — ready to implement.
```

Only when every skeleton is confirmed red does MDD proceed. The skeletons are the finish line. You haven't crossed it yet.

The Build Plan (Phase 5)

MDD auto-detects feature size and generates an appropriate build plan. Simple features (fewer than 3 new files, no API, no database) get flat numbered steps. Anything larger gets a block structure with explicit dependency layers.

Each block specifies:

- A unique name you can reference when asking for changes
- An **end-state**, what is compilable and runnable when the block finishes
- A **commit scope**, the conventional commit message for this block
- A **verification command**, the exact command that proves the block is done
- A **handoff**, what the next block expects to exist when this one finishes
- Whether it runs in the main conversation or with parallel agents

The parallelization logic is strict. Before assigning a block to parallel agents, MDD runs two checks. The **file declaration gate** lists every file each agent will write, and any overlap means sequential execution with no exceptions. The **type dependency gate** checks whether any code one block writes gets imported by code another block writes, and if so, the first block runs before the second.

This is what separates parallel execution that actually works from parallel execution that produces mysterious type errors.

Implementation with the Green Gate (Phase 6)

Implementation follows the dependency layers from the build plan. Within each layer, parallel blocks run simultaneously. After each block, MDD runs the Green Gate loop:

```
Green Gate — Block 2 (Services)

Iteration 1: pnpm test:unit -- --grep "orders" && pnpm typecheck
  Failing: createOrder returns undefined instead of order ID
  Root cause: handler not returning db.insertOne result
  Fix applied: capture and return inserted doc

Iteration 2: All tests passing, typecheck clean
  → regression check...

Full suite: pnpm test:unit
  All pre-existing tests passing — no regressions

Block 2 (Services): ✅
```

If a test fails, MDD requires a diagnosis before any fix gets applied. What is the exact error message, file, and line? Which implementation assumption was wrong? Is this a known pattern from CLAUDE.md or project conventions? What is the single targeted fix?

Tests are never modified to make code pass. If a test seems wrong, MDD re-reads the MDD doc first. If the doc seems wrong, it stops and asks you before touching anything.

After 5 failed iterations, MDD stops completely and presents its findings with three options: keep debugging, narrow the scope, or pause and review together. It never silently gives up or starts tweaking tests to get them to green.

Regressions in existing tests count against the current block's iteration budget. They can't be quietly deprioritized.

Integration Verification (Phase 7)

Tests passing is not done. Phase 7 requires actual integration verification against the real environment.

For **backend features**: real HTTP calls, real database state checked with direct queries, documented error cases verified against actual responses.

For **frontend features**: opening the actual page in a browser, checking the network tab for expected API calls and responses, verifying the console is clean.

For **database features**: write verification via direct DB query (not just trusting the insert return value), constraint testing with invalid data, EXPLAIN on primary query patterns.

For **tooling features**: running against a real scenario, verifying output against the documented behavior, confirming no unintended side effects.

MDD's default stance during integration is: **my code is wrong until proven otherwise**. Any external failure, an API being unreachable, missing test data in the DB, a key not being set, is a hypothesis, not an accepted fact, until empirically disproven with evidence from logs and minimal probes.

When integration is verified, MDD updates the feature doc (`status: complete`, `phase: all`), surfaces any new patterns worth adding to CLAUDE.md, and offers a commit/merge workflow.

AUDIT MODE: FINDING WHAT'S WRONG

`/mdd audit` runs a full-project audit against all documented features. It's meant to be run periodically, after a sprint, before a release, when you suspect something has quietly gone sideways.

Parallelized Reading

The audit uses batched parallel Explore agents to read source files for all features at the same time. Batch sizes scale with how many features you have:

- 1-3 features: single agent (or just the main conversation for very small projects)
- 4-6 features: 2 agents
- 7+: 3 agents (max)

Each agent reads its assigned features' source files and returns structured notes. It doesn't write any files. Only the main conversation writes the notes file, after all agents have returned. This prevents partial writes and keeps the output consistent.

Finding Severity Levels

Audit findings are classified by severity:

- 🔴 **CRITICAL**, security issues, credential exposure, broken contracts
- 🟠 **HIGH**, significant bugs, missing tests for documented behaviors, data model mismatches
- 🟡 **MEDIUM**, code quality violations (file too long, function too long, missing TypeScript types)
- 🔵 **LOW**, documentation drift, minor style inconsistencies

The Audit Report

After notes are collected, MDD produces a structured report:

```
🔍 MDD Audit Complete

Findings: 12 total (2 Critical, 4 High, 4 Medium, 2 Low)
Report: .mdd/audits/report-2026-04-20.md

Top issues:
  1. CRITICAL: orders.ts exposes raw database error messages to API responses
  2. HIGH: GET /api/v1/orders/:id has no test coverage
  3. HIGH: User model has email field without uniqueness constraint

Estimated fix time: 4 hours (traditional) → 45 minutes (MDD)

Fix all now? (yes / review report first / fix only critical+high)
```

That "traditional vs MDD" time estimate isn't marketing. The audit report contains enough structured context that Claude can fix most findings directly without you explaining anything. The MDD doc is the spec, the code is what actually exists, the diff between them is the finding, and the fix is closing that gap.

SCAN MODE: DETECTING DRIFT

`/mdd scan` is the lightweight drift detector. Unlike a full audit, it doesn't read source files at all. It uses git log to check whether any files in a feature's `source_files` list have had commits since the feature's `last_synced` date.

A single Explore agent runs all the git checks simultaneously and returns a classification table:

```
🔍 MDD Scan — Drift Report

  ✅ 01-project-scaffolding   — in sync (last synced: 2026-03-15)
  ⚠️  04-content-builder       — DRIFTED (3 commits since 2026-03-01)
                                 Latest: "fix: markdown heading parser"
  ❌ 07-github-pages           — broken reference (docs/index.html not found)
  ❓ 09-integrations           — untracked (no last_synced field)
```

Classifications:

- **in_sync**, files exist, no commits since last sync
- **drifted**, commits found after `last_synced`
- **broken**, one or more source files not found on disk
- **untracked**, no `last_synced` field yet

For drifted features, the recommended action is `/mdd update`, which resynchronizes the doc with the current code.

UPDATE MODE: KEEPING DOCS IN SYNC

`/mdd update` resynchronizes a feature doc after its code has changed.

MDD reads the current source files and compares them against the doc, looking for new functions or exports that aren't in the doc, things the doc still mentions that have been removed or renamed, data model fields that changed, business rules with different validation or new states, and new edge cases visible in error handling.

It presents the delta to you before rewriting anything. It shows exactly which sections need updating and asks for confirmation. Only changed sections get rewritten, existing prose that's still accurate stays exactly as it is.

After rewriting, MDD generates test skeleton entries for any new documented behaviors and appends them to the existing test file, without touching existing test implementations.

REVERSE-ENGINEER MODE: DOCUMENTING EXISTING CODE

`/mdd reverse-engineer [path]` generates MDD documentation from source code that was written before MDD was adopted.

Without an argument, it scans `src/` and cross-references against existing docs to find undocumented files. You choose which ones to document.

With a file path, it reads that file and infers:

- **Purpose**, what does this do and why?
- **Data models**, TypeScript interfaces, types, Zod schemas
- **API routes**, route definitions and their handlers
- **Business rules**, conditional logic, validation, state transitions
- **Dependencies**, what other modules does it import?
- **Edge cases**, error handling patterns, guard clauses

For four or more files, it uses parallel agents to read them simultaneously and returns a synthesized draft.

MDD always discloses the limitations of reverse-engineered docs before saving:

```
⚠️  Reverse-engineer limitations:
   - "Purpose" section is inferred — review business intent carefully
   - Implicit constraints (SLAs, compliance, product decisions) are not capture
   - Confirm accuracy before treating this doc as the source of truth
```

Business intent isn't in the code. The code tells you what something does, not why it exists or what tradeoffs were made when it was built. Reverse-engineered docs need a human to review the purpose section before they become authoritative.

GRAPH MODE: UNDERSTANDING FEATURE DEPENDENCIES

`/mdd graph` renders the dependency map of all your documented features.

```
📊 MDD Dependency Graph

  06-command-system ──────────────────► 01-project-scaffolding
  09-integrations ────────────────────► 06-command-system
  04-content-builder ─────────────────► 03-database-layer
  05-testing-framework ───────────────► 03-database-layer

Orphans (no dependencies, no dependents):
  07-github-pages
  08-quality-gates

Issues:
  ⚠️  09-integrations depends on 06-command-system (status: in_progress) — risk
  ❌ 05-testing-framework depends on 10-mdd-refinements (deprecated) — broken
```

MDD automatically detects several classes of dependency problems:

- **Broken dependency**, a doc depends on a deprecated or archived feature
- **Risky dependency**, a complete feature depends on a draft or in-progress feature
- **Task dependency**, a feature doc lists a task doc in `depends_on` (tasks are one-off and frozen, they don't carry ongoing contracts)
- **Orphan**, a feature with no dependencies and nothing depending on it

When initiatives and waves are present, the graph also renders the initiative, wave, and feature hierarchy and flags broken links within it.

STATUS MODE: THE AT-A-GLANCE DASHBOARD

`/mdd status` gives you an immediate snapshot of project state:

```
📊 MDD Status

Feature docs:     17 files in .mdd/docs/
Ops runbooks:     2 files in .mdd/ops/
Last audit:       2026-03-01 (20 findings, 17 fixed, 3 open)
Test coverage:    89 unit tests, 12 E2E tests
Known issues:     3 tracked across 2 features
Quality gates:    0 files over 300 lines

MDD version:      v4 — all files up to date

Drift check:
  15 features in sync
  2 features possibly drifted  ← run /mdd scan for details
  0 features untracked
```

After presenting the status, MDD rebuilds `.mdd/.startup.md` with fresh data. That file is what gets injected into future Claude sessions, so every `/mdd status` run keeps the injected context current.

NOTE MODE: A SCRATCH PAD THAT PERSISTS

`/mdd note "your observation here"` appends a timestamped note to the Notes section of `.mdd/.startup.md`:

```
- [2026-04-20] Stripe webhook signature validation needs to happen before JSON
- [2026-04-20] The orders collection needs a compound index on userId + created
```

Because these notes live in `.startup.md`, they get injected into every future Claude session automatically. This is the escape hatch for context that doesn't belong in a feature doc but is too important to lose: performance observations, known gotchas, architectural reminders, anything you want Claude to remember the next time you work in this area.

`/mdd note list` shows all current notes. `/mdd note clear` wipes them with confirmation.

INITIATIVE AND WAVE PLANNING

For larger projects, MDD supports multi-wave initiatives, structured release planning that ties feature docs into a coherent delivery sequence.

Initiatives

An initiative is a named goal with a defined set of waves. Each wave has a **demo-state**, a concrete description of what you should be able to show a user when the wave is complete. Not a task list, a capability milestone.

```
/mdd plan-initiative
```

MDD asks what the initiative is, what it delivers, roughly how many waves it needs, and for each wave, what can the user actually DO when it's done.

The resulting initiative file includes an `Open Product Questions` section. These are unchecked items that block wave planning entirely. You cannot plan or execute a wave until all product questions are answered. This prevents the very common pattern of starting to build before you've resolved the ambiguity that would have changed the architecture.

Waves

```
/mdd plan-wave auth-system-wave-1
```

Each wave contains a feature table with dependency ordering within the wave. MDD enforces a dependency graph check at wave creation time. If your feature ordering would build something before its dependency exists, it stops and offers to auto-reorder.

Executing a Wave

```
/mdd plan-execute auth-system-wave-1
```

You choose between interactive mode (full MDD gates at each feature) or automated mode (minimal interruptions, pauses only on errors or 5-iteration failures).

MDD works through features in dependency order and updates the wave doc as each one completes. If you stop mid-wave and come back later, MDD reads each feature's `wave_status` and resumes at the first incomplete feature. No manual tracking needed.

When all features are complete, MDD shows the demo-state and asks you to verify it manually before marking the wave done. No wave is ever marked complete by a test pass alone.

Integrity Enforcement

Every initiative and wave file has a `hash` field computed from its content. When you run `plan-execute` or `plan-wave`, MDD computes the current hash and compares it to the stored one. If they don't match, it stops:

```
Initiative file has been manually edited since last sync.
Run /mdd plan-sync first.
```

`/mdd plan-sync` reconciles manual edits. It detects which files changed, shows you a diff, and updates version numbers and hashes. If an initiative's version increments, any waves that completed against the old version get flagged for review. Manual edits can introduce inconsistencies, and plan-sync surfaces them before they compound.

OPS MODE: DEPLOYMENT RUNBOOKS THAT EXECUTE

MDD's Ops modes extend the same Document, Execute pattern to deployment operations.

Creating a Runbook

```
/mdd ops deploy swarmk to dokploy
```

MDD asks about services, regions, deployment strategies, gate conditions, rollback procedures, and required credentials (names only, never values). It writes a structured runbook to `.mdd/ops/.md`.

Runbooks can be **project-scoped** (in `.mdd/ops/`) or **global** (in `~/.claude/ops/`). Global runbooks work across all your projects, but they can't access project-local `.env` variables. MDD enforces no slug collision between project and global runbooks.

Executing a Runbook

```
/mdd runop swarmk-dokploy
```

MDD runs the full deployment sequence:

1. **Pre-flight health check**, hits each service's health endpoint in each region and shows a status table
2. **Region-by-region deployment**, deploys in `deploy_order` sequence (canary first, then primary)
3. **Gate check after each region**, runs health checks and applies the configured gate strategy (`health_check`, `manual`, or `none`)
4. **On gate failure**, applies the configured failure strategy (`stop`, `skip_region`, or `rollback`)
5. **Post-flight health check**, full cross-region before and after comparison

This turns a deployment into a verified, documented, repeatable operation. No more "I think I deployed it right." The runbook is the procedure, the execution is tracked, and the health checks tell you exactly what state everything is in before and after.

UPGRADE MODE: MIGRATING EXISTING DOCS

`/mdd upgrade` handles projects that started using MDD before certain frontmatter fields were introduced. It scans all docs, identifies missing fields (`last_synced`, `status`, `phase`), infers sensible defaults from git history, and presents a patch plan for confirmation before writing anything.

```
📋 Upgrade Inventory

Doc                          | last_synced | status | phase
01-project-scaffolding       | ❌ missing  | ❌     | ❌
03-database-layer            | ✅ present  | ✅     | ✅

Docs needing upgrade: 1 of 2
```

After upgrade, `/mdd scan` will have accurate drift data for all docs.

WHY MDD CHANGES THE DEVELOPMENT EQUATION

The benefits stack up in ways that aren't obvious until you've been using MDD for a few weeks.

Context That Survives Session Boundaries

`.mdd/.startup.md` gets injected into every new Claude session. Claude doesn't start blind. It knows what features exist, what their status is, when the last audit ran, what findings are still open. You stop spending the first five minutes of every session re-establishing context you already established last week.

A Spec Claude Can Be Held To

Without docs, Claude is making implicit assumptions about what "correct" means. When something goes wrong, the conversation becomes a negotiation about what the code was supposed to do. With MDD docs, there's no negotiation. The doc says what the API returns. The code either matches or it doesn't. Audits become factual rather than argumentative.

Test Failures That Mean Something

The Red Gate ensures that every test skeleton started as a genuine failure. The Green Gate ensures that every fix is targeted and diagnosed rather than just "change things until it goes green." And regression checks after each block mean you catch breakage at the block where it was introduced, not three blocks later when you have no idea what changed.

Changes That Can Be Explained

When someone asks "why does this code work this way?", the MDD doc has the answer. It was designed to handle the edge cases listed in Business Rules. It depends on the contract defined in `03-database-layer`. It was built to satisfy the requirements captured in Phase 1. You have a paper trail that didn't exist before.

Drift Detection Before It Becomes a Bug

`/mdd scan` takes about 30 seconds and tells you which documented features have had code changes since their last MDD session. Run it before a release and you know exactly which features might have deviated from their spec. Wire it into CI and drift becomes a first-class metric instead of an invisible accumulating problem.

Deployments That Are Repeatable

Ops runbooks aren't just documentation, they execute. Every deployment follows the same pre-flight, region-sequenced, gate-checked procedure. Rollback steps are defined before the deployment starts, not improvised at 2am after something went wrong in production.

GETTING STARTED

The MDD workflow ships with the Claude Code Mastery Project Starter Kit. If you're already using the starter kit, you have MDD available right now. Run `/mdd` to start your first feature.

If you're not using the starter kit, you can install MDD into any existing project:

```
/install-mdd [path]
```

This copies the MDD command and scaffolds the `.mdd/` directory structure. It's non-destructive, nothing gets overwritten.

For existing codebases with no documentation, start with:

```
/mdd reverse-engineer
```

MDD will scan your source files, find undocumented code, and help you create documentation from the existing implementation. Once your major features are documented, run `/mdd audit` to find gaps, then `/mdd scan` to track drift going forward.

THE MENTAL MODEL SHIFT

The shift MDD asks you to make is accepting that documentation is not waste. It is the asset. Code is ephemeral. Requirements change, refactors happen, files get renamed, features get removed. The documentation captures what was intended, what was decided, what was deferred, and why. That intent is exactly what you lose when you "just use Claude" and close the tab.

MDD is how you make Claude a reliable long-term collaborator instead of a capable short-term assistant. The investment in documentation pays off every time you open a new session and Claude picks up exactly where you left off, every time an audit catches a real bug before it reaches production, every time a scan tells you something has drifted before it breaks something downstream.

The workflow is built for the way serious software development actually works: across many sessions, across many features, with real dependencies, real constraints, and real consequences when things go wrong. That's the environment where MDD earns its keep.