Markdown for AI

Why static markdown fails AI workflows, and what MarkdownAI does instead

MAY 17, 2026

THEDECIPHERIST.COM

TABLE OF CONTENTS

The numbers below come from rewriting MDD - a real AI development workflow tool - from plain Markdown to MarkdownAI. Same tool. Same tasks. Same scenarios.

	PLAIN MARKDOWN	MARKDOWNAI	
Total workflow size	~32,000 tokens	~14,500 tokens	-55%
Greenfield build session	3 min	1.3 min	57% faster
Brownfield audit session	5.4 min	under 1 min	86% faster
Multi-mode session (4 modes)	12.7 min	2.4 min	81% faster
Claude tool calls per session	21	0	eliminated
Mid-session compaction	frequent	none	eliminated
Shared logic	copy-pasted 5 times	defined once	-

Three things are broken about the way AI workflows use markdown files today. Not because of the AI. Because of the format.

Docs go stale the moment you finish writing them. Your AI reads them like they were written this morning. They were not. The port changed, the schema evolved, the env var got renamed. The markdown stayed frozen. The AI proceeds on bad information and does not know it.

You are paying Claude to do work a shell script does for free. Every time your AI checks what branch you are on, counts your files, or verifies whether a config exists - that is your context window and your token budget going to something computers have handled for decades. There is no reasoning required. A script does it in milliseconds. Claude does it slowly, in your session, at a cost.

The same logic lives in five files because markdown has no macros. So you copy-paste the same branch check, the same startup instructions, the same boilerplate into every command file. Claude reads every copy on every session. Define it once. Call it everywhere. Claude does not need to see the repetition.

MarkdownAI solves all three. One line at the top of any `.md` file.

THE PROBLEM IS ARCHITECTURAL

Markdown was designed in 2004 to format static text for human readers on the web. It does that job well. But there is a fundamental mismatch between what Markdown is and what AI workflows need from it.

A human reader brings prior knowledge to a document. When something looks off they check. They cross-reference. They update their mental model. An AI assistant reads what is in the file and treats it as current fact. It cannot know whether the information is ten minutes old or ten months old. It reads, trusts, and proceeds - confidently wrong.

This compounds in agentic workflows. When an AI is navigating a multi-step process - reading documentation at each phase, making decisions based on what it reads, then acting - stale documentation does not just produce one wrong answer. It contaminates the session. By the time the error surfaces, the agent has made five downstream decisions on bad premises.

Standard Markdown has no answer to this. A `.md` file stores what you write. The gap between what was written and what is true right now grows continuously, and there is nothing in the format to prevent it.

A DIFFERENT KIND OF DOCUMENT

MarkdownAI's premise is simple: a document should not store facts. It should fetch them.

Add `@markdownai` to the first line of any `.md` file:

```
@markdownai

# Project Status

**Branch:** {{ @query git branch --show-current }}
**Tests:** {{ @query pnpm test 2>&1 | tail -1 }}
**Last commit:** {{ @query git log --oneline -1 }}
**Open PRs:** {{ @query gh pr list --json number | jq '. | length' }}

@if env.NODE_ENV == "production"
**Environment: PRODUCTION**
@else
**Environment:** {{ env.NODE_ENV }}
@endif
```

Run `mai render status.md`. The directives execute. Every `{{ @query }}` runs the actual command. Every `@if` evaluates against the real environment. The output is clean, standard Markdown with accurate data from your system as it exists right now.

The file has a `.md` extension. It renders as normal Markdown to any standard tool. Files without `@markdownai` are completely unaffected. You add it where accuracy matters and leave everything else alone.

WHAT THIS CHANGES FOR AI WORKFLOWS

The immediate benefit - accurate documentation - is obvious. The more interesting change is what becomes possible once you stop handing deterministic work to a reasoning engine.

The AI stops running commands it should never have needed to run

In a standard AI workflow, the AI runs diagnostic checks because no one has run them yet. With MarkdownAI, those queries run during document rendering - before the session starts. By the time Claude reads the workflow file, the branch name is already in the document. The file counts are already there. The config check already ran. Claude receives a document with the answers pre-filled and spends the session on things that actually require thought.

Here is what this looks like in practice. MDD is a document-driven development system for Claude Code. After being rewritten to use MarkdownAI:

- **Before:** 21 tool calls Claude had to make per build session just for diagnostic checks
- **After:** zero Claude tool calls - all queries run at render time in milliseconds

Each of those tool calls had a round-trip cost, plus Claude's reasoning time about what the result meant. That overhead used to compound across every session.

Conditional stripping keeps context windows honest

MarkdownAI evaluates `@if` blocks before the document reaches Claude. Content in branches that do not apply is stripped. Claude never sees it.

For a greenfield project with no existing feature docs, the brownfield analysis sections do not load:

```
@call doc-count                    => doc_count = "0"

@if {{ doc_count }} == "0"
@constraint Skip Phase 2 - greenfield. No existing docs. @end
@else
[full brownfield analysis instructions]  <- stripped
@endif
```

In an audit session with no stale job state, the resume logic does not load. Every session gets exactly the context it needs - nothing more.

Write it once. Call it everywhere.

Standard markdown means copy-pasting shared logic across every file that needs it. The same branch safety check, the same startup rebuild steps, the same connection instructions - duplicated every time.

```
@define branch-guard
@query bash -c "git branch --show-current" label=current_branch
@query bash -c "git status --porcelain" label=dirty_check
[...branch safety logic using live values...]
@end
```

Define it once in a shared file. Call it in every command file that needs it:

```
@import mdd-shared.md
@call branch-guard
```

Claude sees the result - the branch name already filled in, the check already evaluated. It never sees the definition. It never reads the same block five times. You maintain one copy and every file that imports it stays in sync automatically.

The numbers from a real rewrite

MDD was rewritten from plain Markdown command files to MarkdownAI-native command files. Three session types measured:

Token reduction:

SESSION TYPE	BEFORE	AFTER	REDUCTION
Greenfield build	9,003	7,641	20%
Brownfield audit	7,040	4,075	50%
Multi-mode (4 modes)	22,057	14,584	40%

Session time reduction:

SESSION TYPE	BEFORE	AFTER	FASTER
Greenfield build	3 min	1.3 min	57%
Brownfield audit	5.4 min	under a minute	86%
Multi-mode workflow	12.7 min	2.4 min	81%

The audit scenario is the most dramatic. The earlier phase findings need to stay in context while later remediation is written. When there is too much overhead, the context compacts and Claude loses what it found earlier. The audit appears to

complete but the remediation was written without the full analysis in context. The token savings v2 gives you is the margin that prevents this.

The 86% time reduction in the audit scenario is not mostly from fewer tokens. It is from eliminating the correction loops - the interruptions where you have to stop and explain to Claude that what it just said confidently stopped being true months ago.

HOW THE INTEGRATION WORKS

Zero workflow change

Run `mai init`. This installs a PreToolUse hook into Claude Code or Cursor. From that point, when Claude reads any `.md` file that starts with `@markdownai`, the hook intercepts the read and routes it through `mai render` first. Claude receives executed, live output.

Your workflow does not change. Claude still reads `.md` files the same way it always did. The files just start being accurate.

MCP for programmatic control

`mai serve` starts an MCP server. Claude can navigate `@phase` workflows programmatically - checking whether a phase's preconditions are met before executing it, pulling structured `@constraint` rules out of a document, executing individual macros for targeted data. The MCP approach is the right fit when you are building autonomous agents that need to verify preconditions rather than just read instructions and hope they were followed.

DESIGNING AI WORKFLOWS DIFFERENTLY

Once you have live documents, you can design workflows that were not practical before.

Pre-flight checks that actually run

Standard AI runbooks describe steps: "check that the health endpoint returns 200," "verify the environment variable is set." The AI reads these and runs them as tool calls.

With MarkdownAI `@phase` blocks, the verification runs before Claude reaches that section:

```
@phase pre-flight
@env DEPLOY_TOKEN required
@http GET {{ env.HEALTH_ENDPOINT }}/health expected=200
@http GET {{ env.DB_ENDPOINT }}/ready expected=200

@on complete
  Pre-flight passed. All services healthy.
@end

@phase deploy
@query bash -c "kubectl set image deployment/{{ env.APP_NAME }} {{ env.IMAGE }}
@end
```

The pre-flight phase fails if the health endpoint does not return 200. The deploy phase does not start. This is verification that actually happens, not verification that Claude is told to do.

Status documents that update themselves

```
@markdownai

# Release Status - {{ @date format="YYYY-MM-DD HH:mm" }}

**Branch:** {{ @query git branch --show-current }}
**Tests:** {{ @query pnpm test 2>&1 | tail -1 }}

## Pending Changes
@query bash -c "git log main..HEAD --oneline" | @render type="numbered"
```

Add `mai render status.md -o status-rendered.md` to your CI pipeline. Every run produces an accurate status document. No one has to update it. It cannot be stale.

THE ADOPTION PATH

You do not need to convert anything to get started. Files without `@markdownai` are completely unaffected. Add it to the files where accuracy matters most - your AI workflow instruction files, your status documents, your runbooks - and leave everything else alone.

1. Install `mai` globally: `npm install -g @markdownai/core`
2. Install the hook: `mai init`
3. Add `@markdownai` to one workflow file
4. Add one live directive: `{{ @query git branch --show-current }}`
5. Run `mai render`. See what accurate looks like.

The full feature set is there when you need it. You do not have to adopt it all at once.

WHAT CHANGES WHEN YOU BUILD THIS WAY

The productivity numbers are real, but they are a side effect. The more significant change is in how you design AI workflows once you stop treating the `.md` file as a static brief.

When a document can query your system, you start writing documents that describe current reality rather than hoped-for reality. Your runbook does not say "make sure the health endpoint is up" - it checks it and fails if it is not. Your workflow file does not say "you are probably on a feature branch" - it knows which branch and has already evaluated whether it matches the task. Your status doc does not say "we have roughly X records" - it ran the query.

The documents become the source of truth instead of a copy of it. That is a fundamentally different relationship between your team, your tools, and the AI that works with both.

```
npm install -g @markdownai/core
mai init
```

689 tests, full CLI, MCP server, PreToolUse hook. MIT license.

GitHub

- TheDecipherist/markdownai

npm

- `@markdownai/core` - CLI (`mai` binary) - start here
- `@markdownai/mcp` - MCP server for Claude / Cursor integration
- `@markdownai/parser` - AST parser (inert, no execution)
- `@markdownai/engine` - directive execution engine
- `@markdownai/renderer` - 11 output format modules
- `@markdownai/markdownai` - meta package (installs all of the above)