ENGINEERING GUIDE

Working with AI Coding Agents

Agent configuration, enforcement layers, and the logging standards that make LLM-generated code debuggable—grounded in what actually works as of early 2026.

Audience: Developers & technical leads **Updated:** February 2026 **Reading time:** 25 min

CONTENTS

01

What Are AI Coding Agents?

An AI coding agent is an LLM-powered tool that can read your codebase, write code, run shell commands, and make multi-step decisions to accomplish a task. Unlike autocomplete (which predicts your next line), agents operate autonomously across files and execute sequences of actions.

The key distinction: **autocomplete reacts to your cursor; agents pursue goals.**

TOOL	TYPE	HOW IT WORKS
Claude Code	CLI	Terminal-based; filesystem access, shell commands, hooks, sub-agents, skills
Cursor	IDE	VS Code fork with agent mode; reads codebase, edits files, runs terminal
GitHub Copilot	IDE ext.	Autocomplete + agent mode for multi-file edits and PR creation
OpenAI Codex	Cloud	Sandboxed; operates on branches asynchronously
Windsurf	IDE	Cascade agent indexes entire codebase; enterprise focus
Zed + Aider	Editor / CLI	Lightweight agents with AGENTS.md support

KEY INSIGHT FOR NEWCOMERS

These agents are **stateless between sessions**. Every new conversation starts from scratch. The agent doesn't remember yesterday's discussion, your preferences, or past decisions. This is why configuration files exist—they inject persistent context that survives between sessions.

02

Why Agents Need Rules (and Why Rules Aren't Enough)

Without explicit rules, an agent will guess your tech stack (often wrong), pick its own code style, generate minimal logging, explore your codebase from scratch every time, and make architecture decisions that contradict established patterns.

Agent rules solve this by injecting your project's context at the start of every session. Think of them as onboarding documentation for an AI.

THE REAL-WORLD FAILURE MODE

An agent generates a Python script that processes credit applications. Without rules, it produces code with `print("done")` at the end. Six months later, when something fails at 2 AM, the person debugging has no idea what the code did, which application failed, or why.

But here's the part most guides miss: even *with* rules, the agent might still skip your logging standard. Rules in config files are **advisory**—the LLM reads them but can still ignore them under context pressure. For critical standards, you need enforcement beyond config files. That's what Section 8 covers.

03

The Configuration Landscape

Each tool invented its own config file format. The good news: this fragmentation is resolving rapidly. As of late 2025, most major tools read `AGENTS.md` natively or via symlinks.

TOOL	NATIVE FILE	AGENTS.MD SUPPORT
Claude Code	`CLAUDE.md`	Reads if present (symlink recommended)
Cursor	`.cursor/rules/*.mdc`	Native since late 2025
GitHub Copilot	`.github/copilot-instructions.md`	Native
OpenAI Codex	`AGENTS.md`	Native (originator)
Windsurf	`.windsurfrules`	Symlink
Zed, Aider, OpenCode	Various	Native

PRACTICAL TAKEAWAY

Use `AGENTS.md` as your single source of truth and symlink to tool-specific files. But know this: **Cursor's glob-based auto-activation** in `.cursor/rules/` and **Claude Code's hooks and skills** offer capabilities that `AGENTS.md` alone cannot replicate. You may still need tool-specific files for advanced features.

04

AGENTS.md — The Emerging Standard

`AGENTS.md` is an open standard stewarded by the **Agentic AI Foundation** under the Linux Foundation. It was released by OpenAI in August 2025 and has been adopted by over 60,000 open-source projects. The AAIF was co-founded in December 2025 by OpenAI, Anthropic, Block, AWS, Google, Microsoft, Bloomberg, and Cloudflare.

How it works:

- Place an `AGENTS.md` at your project root—plain Markdown, no special schema
- Agents read the nearest `AGENTS.md` in the directory tree (for monorepo support, nest them)
- User prompts always override the file contents

```
# Set up AGENTS.md as single source of truth:

# Claude Code
ln -s AGENTS.md CLAUDE.md
```

```
# Windsurf
ln -s AGENTS.md .windsurfrules

# GitHub Copilot
mkdir -p .github
ln -s ../AGENTS.md .github/copilot-instructions.md

# Cursor reads AGENTS.md natively (late 2025+)
# For older versions:
ln -s AGENTS.md .cursorrules
```

05

What Goes in a Config File

Not all rules are created equal. Research and practitioner experience consistently show that **concrete, verifiable instructions** outperform abstract guidance. The order below reflects measured impact.

CONTEXT BUDGET WARNING

Research from HumanLayer shows that Claude Code's system prompt alone contains ~50 individual instructions. Frontier reasoning models can follow roughly 150–200 instructions before degradation becomes significant. As instruction count rises, compliance degrades **uniformly**—the model doesn't just forget later items, it starts ignoring everything more. Keep your config file under 150 lines.

Tier 1: High Impact (Always Include)

Project Overview — Two or three sentences: what the project does, core tech stack, and purpose. This prevents the agent from exploring your codebase blindly.

Commands — Exact commands for build, test, lint, and deploy. Write them as the agent should run them, inside code blocks. Don't write "install dependencies"—write `pnpm install` .

Code Style & Conventions — Specific patterns: functional vs. class-based, naming conventions, error handling. **Point to real files as examples** rather than describing patterns abstractly. "Use functional components like `Projects.tsx` ; avoid class-based patterns like `Admin.tsx` " beats a paragraph of explanation.

Logging Standards — The single most impactful rule for long-term maintainability (see Section 7).

Tier 2: Medium Impact

Architecture Pointers — Describe capabilities, not file paths. "Authentication logic is in the auth module" is better than "Auth is at `src/modules/auth/handlers.ts` "—paths change, capabilities don't. Stale paths actively poison the agent's context.

Dos and Don'ts — Explicit patterns to follow and avoid, with file references.

Testing Requirements — Framework, where tests live, whether to write tests first.

Tier 3: Diminishing Returns

PR/Commit Guidelines, **Security Notes**, **Persona Instructions** — Simple persona labels like "you are a senior engineer" have minimal measured impact. However, domain-framing context ("this is a regulated financial services codebase where auditability matters") can meaningfully shape output for domain-specific tasks.

06

The Enforcement Pyramid

This is the concept most existing guides miss entirely. Not all rules should be enforced the same way. Think of enforcement as a pyramid with three layers:

Enforce

Hooks & CI Gates

Deterministic. Shell commands that run at lifecycle points. The agent cannot bypass these. Exit code 2 = blocked, no negotiation. Use for: formatting, linting, test execution, security scanning, blocking writes to sensitive files.

Advise

Config Files (AGENTS.md / CLAUDE.md)

Advisory. The agent reads these and follows them most of the time, but under context pressure or conflicting signals, it may deviate. Use for: code style, architecture guidance, logging patterns, naming conventions.

Suggest

In-Prompt Guidance

Ephemeral. Contextual instructions in your prompt. Highest priority for the current task but gone next session. Use for: one-off requirements, task-specific constraints.

THE PRACTICAL IMPLICATION

If a rule **must** be followed every single time (like running `prettier` after edits, or never writing to `.env`), put it in a hook. If a rule **should** be followed but occasional deviation is acceptable (like logging patterns), put it in your config file. If it only applies to this task, put it in your prompt.

07

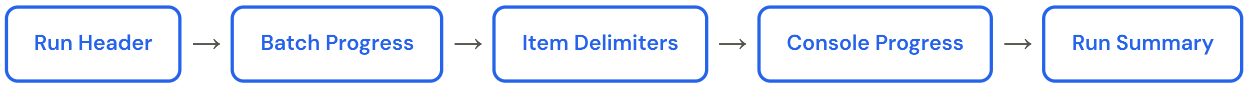
Logging Standards — The Highest-ROI Rule

Logging is the single most impactful standard you can enforce through agent rules, for several compounding reasons:

- **LLMs naturally underlog.** They optimize for "it works" not "it's observable." Without rules, you get `print("processing...")` and nothing else.
- **Most code is now LLM-generated.** The person debugging at 2 AM probably didn't write the code. Logs are their only window.
- **It applies universally** regardless of language, framework, or project type.
- **It's verifiable.** Unlike "write clean code," logging patterns are concrete and greppable.

The Five-Part Logging Pattern

When the agent writes code that processes collections (records, API calls, files, jobs—anything in a loop or batch), it should implement:



PART 1: RUN HEADER

Log what's about to happen—total items, filtered count, filters applied, batch size.

```
========================================
RUN STARTED: 2026-02-14 09:30:15
Total items found: 150
Items matching filters: 42
Filters applied: status=pending, date≥2026-01-01
Batch size: 10 | Estimated batches: 5
========================================
```

When you open this log, you immediately know scope. If 150 items exist but only 42 processed, the filter is responsible—not a crash.

PART 2: BATCH PROGRESS

```
--- Batch 1/5 (items 1-10 of 42) ---
--- Batch 2/5 (items 11-20 of 42) ---
```

PART 3: ITEM DELIMITERS (FILE LOG)

Wrap each item between markers that are both human-scannable and machine-parseable:

```
>>>>>> ITEM START [14/42] id=ENV-2026-0042 at 2026-02-14 09:30:17 <<<<<<
  Validating envelope ENV-2026-0042 against schema v2.1
  Schema validation passed
  Sending to eKYC provider Evrotrust (attempt 1/3)
  eKYC response received: status=verified, confidence=0.94
  Updating database record: status → VERIFIED
<<<<<< ITEM END [14/42] id=ENV-2026-0042 result=SUCCESS duration=1.23s >>>>>>
```

Rules: use exactly six angle brackets (unique enough to grep reliably). Include sequence number, item ID, and timestamp on START. Include result status and duration on END. Indent everything between START and END.

PART 4: CONSOLE OUTPUT (ONE LINE PER ITEM)

```
[1/42]  Processing ENV-2026-0042... OK (1.23s)
[2/42]  Processing ENV-2026-0043... FAILED: Certificate expired
[3/42]  Processing ENV-2026-0044... SKIPPED: Already verified
```

PART 5: RUN SUMMARY

```
========================================
RUN COMPLETED: 2026-02-14 09:35:22
Total duration: 5m 7s
Processed: 42/42
  Successful: 38
  Failed: 3
  Skipped: 1
Failed items: ENV-2026-0042, ENV-2026-0051, ENV-2026-0089
========================================
```

Log Message Quality

Since the code is LLM-generated, the person reading logs likely didn't write the code. Every message must be self-explanatory:

Bad: Opaque

```
logger.info("Step 3 done")
logger.info(f"Processing {id}")
logger.error("Failed")
logger.info("Retrying...")
```

Good: Self-Explanatory

```
logger.info("Validating ENV-042 against schema v2.1")
logger.info("Sending ENV-042 to eKYC (attempt 2/3)")
logger.error("ENV-042: cert expired 2026-01-15")
logger.info("Retrying eKYC for ENV-042 in 5s (3/3)")
```

When to Use Structured JSON Logging Instead

The human-readable pattern above is ideal for scripts, batch jobs, and CLI tools. For production services feeding into observability stacks (ELK, Datadog, Grafana), consider specifying **structured**

JSON logging in your config:

```
# In your AGENTS.md, add:
For production services, emit structured JSON logs:
{"ts":"...","level":"info","op":"ekyc_verify","item_id":"ENV-042","attempt":2,"max":3}

For scripts and CLI tools, use the human-readable
ITEM START/END delimiter pattern described above.
```

Both patterns serve the same goal—observable code. The format depends on what will consume the logs.

Useful Commands for Log Analysis

```
# Find all failures
grep "ITEM END.*FAILED" run.log

# Extract failed item IDs
grep "ITEM END.*FAILED" run.log | grep -oP 'id=\K[^ ]+'

# Show everything for a specific item
sed -n '/ITEM START.*id=ENV-2026-0042/,/ITEM END.*id=ENV-2026-0042/p' run.log

# Count results by type
grep -c "result=SUCCESS" run.log
grep -c "result=FAILED" run.log
```

08

Hooks: When Rules Must Be Guarantees

This section addresses the biggest gap in most AI coding guides. Config file rules are **advisory**—the LLM reads them, understands them, and can still ignore them. As one practitioner documented: "You can write 'NEVER edit .env files' in your CLAUDE.md. Claude will read it. Claude will understand it. And Claude might still edit your .env file."

Hooks are shell commands that execute at specific lifecycle points in Claude Code. They are **deterministic**—they run every time the matching event fires, regardless of what the model thinks it should do.

Hook Events That Matter Most

EVENT	WHEN IT FIRES	BEST USE
`PostToolUse`	After a file edit/write	Auto-format with prettier/gofmt, run linter

EVENT	WHEN IT FIRES	BEST USE
`PreToolUse`	Before a tool runs	Block writes to `.env`, prevent `rm -rf` footguns
`Stop`	When the agent finishes	Run test suite, type-check, security scan
`SessionStart`	Session begins	Inject dynamic context (recent commits, open tickets)
`PermissionRequest`	Agent asks for permission	Auto-approve safe ops, block destructive ones

Example: Auto-Format After Every Edit

```
// .claude/settings.json
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [
          {
            "type": "command",
            "command": "jq -r '.tool_input.file_path' | xargs npx prettier --write"
          }
        ]
      }
    ]
  }
}
```

Example: Block Writes to Sensitive Files

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [
          {
            "type": "command",
            "command": "jq -r '.tool_input.file_path' | grep -qE '\\.(env|pem|key)$' && exit 2
          }
        ]
      }
    ]
  }
}
```

Exit code `2` = action blocked. The agent cannot negotiate its way past this.

> THE COMBINED STRATEGY
>
> **Config files** teach the agent what good looks like. **Hooks** guarantee critical rules are enforced. **Linters and CI** catch everything else. Use all three layers—don't rely on any single one.

09

Honest Assessment: What Works, What's Hype

Genuinely Works

PRACTICE	WHY
Exact commands in code blocks	Agents copy-paste instead of guessing. Universally recommended.
Real files as examples	Concrete examples beat abstract descriptions for LLMs. Aligns with few-shot prompting research.
Short config files (<150 lines)	Instruction-following degrades uniformly as count rises.
Logging rules	LLMs consistently underlog without them. Every team that adds them reports improvement.
Hooks for critical rules	Deterministic. Can't be ignored. Catches what advisory rules miss.
Deferring to linters	Instead of duplicating formatting rules in config, tell the agent to run the linter. Single source of truth.

Has Caveats

PRACTICE	THE CLAIM	THE REALITY
Detailed file paths	"Helps navigation"	Paths go stale quickly. Stale paths actively mislead. Describe capabilities instead.
Simple persona labels	"You are a senior engineer"	Minimal measured impact. But **domain-framing context** ("regulated fintech codebase") can meaningfully shape output.
Complex multi-file rules	"Modular rules for every concern"	Maintenance burden grows fast. Rules go out of sync. One well-maintained file beats 15 scattered ones for most teams.

PRACTICE	THE CLAIM	THE REALITY
AGENTS.md as "universal"	"One file for all tools"	Symlinks work, but Cursor's glob-activation and Claude Code's hooks/skills offer capabilities it can't replicate.

Overhyped

CLAIM	WHY
"Agent observability platforms are essential"	LangSmith, Arize, etc. are for AI systems in production (chatbots, RAG). For coding agents, structured logging in generated code is more valuable than instrumenting the agent itself.
"AI agents will replace developers"	Agents are productivity multipliers. Every output needs human review. The review is never zero.
"Different rule sets per language"	Logging, architecture, and testing rules are language-agnostic. Language-specific concerns (linters, import styles) are better handled by actual linters.

THE 80/20 RULE

Most of the value comes from a small set of concrete rules: logging patterns, exact build/test commands, 5–10 dos/don'ts, and hooks for critical guardrails. Everything beyond that has diminishing returns and increasing maintenance cost. **The biggest risk isn't too few rules—it's stale rules that actively mislead.**

10

Implementation Playbook

Step 1: Create Your AGENTS.md 30 MIN

~~~
# AGENTS.md

## Project
[Project name] is a [brief description]. Built with [tech stack].

## Commands
```
[package-manager] install      # Install dependencies
[package-manager] run dev      # Start dev server (port [N])
[package-manager] run test     # Run all tests
[package-manager] run lint     # Lint and auto-fix
```

## Code Style
~~~

```
- [Your conventions: functional vs OOP, naming, etc.]
- Good example: see `[path/to/exemplary/file]`
- Avoid patterns in: `[path/to/legacy/file]`

## Logging Standards
When writing batch processing or pipeline code, implement
dual logging (file + stdout) with these requirements:
- Run header: total count, filtered count, filters, batch size
- Batch progress: "Batch N/Total (items X-Y of Z)"
- File log delimiters:
  ">>>>>> ITEM START [N/Total] id=ID at TIMESTAMP <<<<<<"
  "<<<<<< ITEM END [N/Total] id=ID result=STATUS duration=Xs >>>>>>"
- Console: one line per item "[N/Total] Processing ID... STATUS"
- Run summary: duration, success/fail/skip counts, failed IDs
All log messages must name the operation, include the item ID,
and be self-explanatory without reading source code.

For production services, use structured JSON logging instead.

## Do / Don't
- DO: [specific instruction with file reference]
- DON'T: [specific antipattern with file reference]

## Testing
- Framework: [Jest/pytest/etc.]
- Run single test: `[command]`
- Tests live in: `[directory]`
```

Step 2: Set Up Symlinks 5 MIN

```
# From project root:
ln -s AGENTS.md CLAUDE.md
ln -s AGENTS.md .windsurfrules
mkdir -p .github
ln -s ../AGENTS.md .github/copilot-instructions.md
```

Step 3: Add Hooks for Critical Rules 15 MIN

Use the interactive `/hooks` command in Claude Code, or create `.claude/settings.json` :

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [{
          "type": "command",
          "command": "jq -r '.tool_input.file_path' | xargs npx prettier --write 2>/dev/null ||
        }]
      }
    ],
    "PreToolUse": [
      {
        "matcher": "Edit|Write",
        "hooks": [{
          "type": "command",
          "command": "jq -r '.tool_input.file_path' | grep -qE '\\.(env|pem|key)$' && exit 2 ||
        }]
      }
    ]
  }
}
```

Step 4: Add Cursor Glob Rules (Optional) 15 MIN

If your team uses Cursor and wants context-aware rule activation, create `.cursor/rules/logging-standards.mdc` :

```
---
description: Logging standards for batch processing and data pipelines.
globs: "**/*.{py,js,ts,cs,java,go,rb,php}"
alwaysApply: false
---

[Paste logging standards from your AGENTS.md here]
```

Step 5: Treat It Like Code ONGOING

- ☐ Update `AGENTS.md` in the same PR when conventions change
- ☐ Add a rule the **second time** you see the agent make the same mistake (not the first—one-off mistakes aren't worth a permanent rule)
- ☐ Delete anything that's no longer true. Stale rules are worse than missing rules.
- ☐ Review quarterly. Delete more than you add.
- ☐ Ask the agent to audit: "Review this AGENTS.md against the current codebase and flag anything stale"

11

Agent Observability — Cutting Through the Confusion

There are two distinct concerns that newcomers routinely conflate:

Observability of Generated Code

Observability of the Agent Itself

Making sure the code the agent writes is observable—proper logging, error handling, metrics. **This is what Section 7 addresses.**

Tools: Your existing logging framework + the rules above. No new infrastructure.

Monitoring the agent's decision-making: tracing reasoning, tracking token usage, evaluating output quality.

Tools: LangSmith, Langfuse, Arize, OpenTelemetry GenAI. Only relevant if building LLM-powered production systems.

For most teams using Cursor or Claude Code to write application code, focus on the first category. The agent observability platforms solve a different problem. If you do build LLM-powered applications (chatbots, RAG pipelines, autonomous workflows), then agent observability becomes relevant—the industry is converging on OpenTelemetry semantic conventions for GenAI.

12

Recommendations

DO THESE SIX THINGS

1. **Create an `AGENTS.md`** at the root of every active project. Keep it under 150 lines. Focus on commands, code style, and logging.
2. **Set up symlinks** to `CLAUDE.md`, `.windsurfrules`, etc. so every tool reads the same source.
3. **Add logging standards** from Section 7. This is the highest-ROI advisory rule you can write.
4. **Add hooks** for anything that must be deterministic: formatting, security boundaries, test execution.
5. **Defer to existing tooling.** Instead of describing formatting rules in your config, tell the agent to run the linter. Single source of truth.
6. **Treat `AGENTS.md` as code**: version it, review it, prune it quarterly. Stale rules cause more harm than missing rules.

SKIP FOR NOW

- Complex multi-file rule hierarchies—start simple, add complexity when you hit specific problems
- Agent observability platforms—unless you're building LLM-powered production systems
- Simple persona labels—concrete rules beat "you are a senior engineer" every time
- Detailed file path documentation—it goes stale and actively misleads

Further Reading: AGENTS.md (official) · How to Write a Good CLAUDE.md · HumanLayer: Writing a Good CLAUDE.md · Claude Code Hooks Guide · Claude Code Best Practices