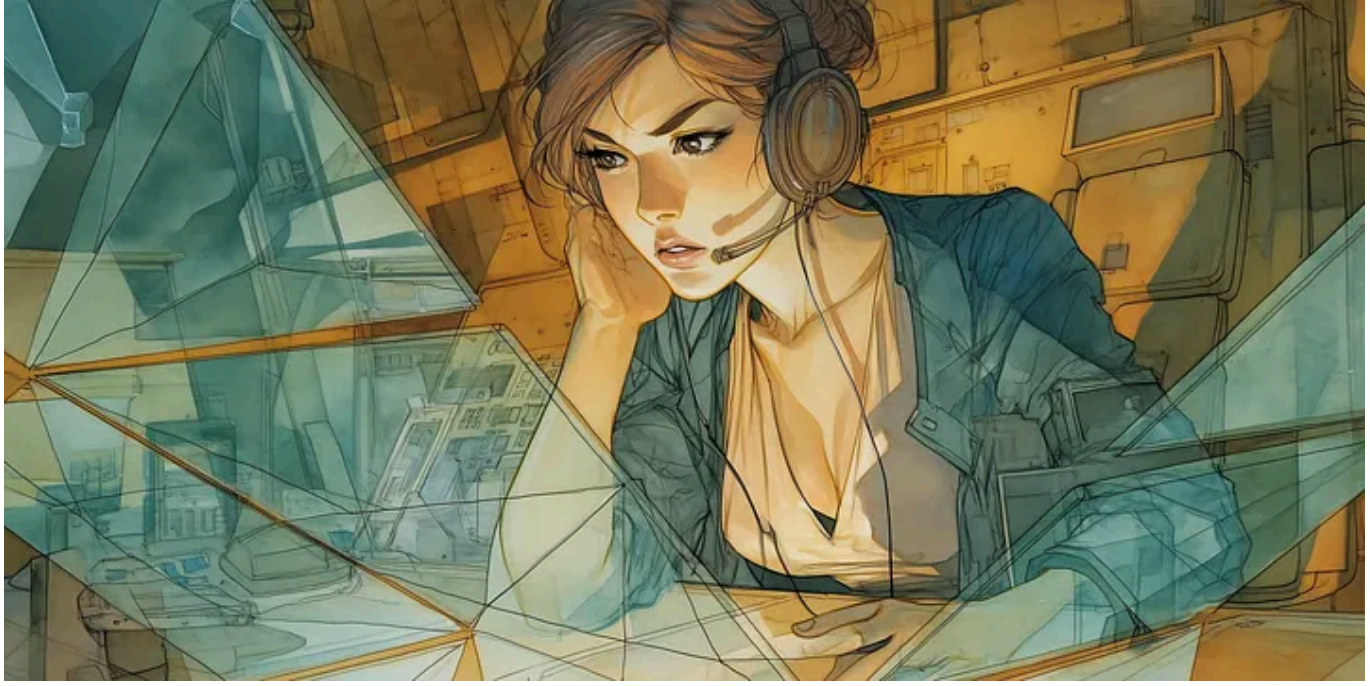
Member-only story

How I Turned Claude Code Into a War Machine (Part 2)

Skills, Hooks, and Commands: The Automation Layer That Completes the Arsenal

Delanoe Pirard Follow 17 min read · Dec 8, 2025

> *"We shape our tools, and thereafter our tools shape us." — Marshall McLuhan*
>
> *Or my 2025 version: We configure Claude, and thereafter Claude configures our way of coding. The question is whether you're doing the shaping — or being shaped by default.*



The War Machine arsenal is complete. Part 2 adds the discipline layer.

. . .

Previously on “War Machine”

In Part 1, I showed you how to build the foundation: **16 expert agents** that route automatically, **6 MCPs** that give Claude access to real data, and an **anti-hallucination protocol** baked into your global CLAUDE.md.

That setup made Claude Code 10x more useful. But it had a flaw.

The agents could verify. The MCPs could fetch. But nothing *enforced* verification.

I was still trusting Claude to remember its instructions. And LLMs have a memory problem.

This article completes the arsenal with the **enforcement layer**: Skills that auto-trigger verification, Hooks that block dangerous actions, and Commands that compress complex workflows into single words.

If Part 1 gave Claude knowledge, Part 2 gives it discipline.

How I Turned Claude Code Into a War Machine (part 1)

From drowning in 1 project to crushing 5 in parallel. Here's the complete playbook.

medium.com

. . .

TL;DR

- **Skills** are model-invoked instruction sets that Claude loads dynamically when relevant — the anti-hallucination layer
- **Hooks** intercept 9 different events with 3 exit codes (0=allow, 1=warn, 2=block) — deterministic control over AI behavior
- **Commands** are semantic shortcuts with powerful frontmatter: `model`, `allowed-tools`, `disable-model-invocation`
- **The architecture:** Skills define *what Claude knows*, Hooks control *what Claude can do*, Commands automate *what you frequently ask*

. . .

The Bug That Cost Me a Weekend

It was 2 AM on a Saturday. I had been debugging a production issue for six hours.

The culprit? A hallucinated API signature. Claude had confidently generated code using `client.stream_chat()` — a method that had never existed in the library I was using. The real method was `client.chat.completions.create(stream=True)`.

Six hours. Because of a confident hallucination.

That weekend, I rebuilt my entire Claude Code configuration from scratch. Not to make Claude faster or more capable — but to make it *honest*. To force verification before generation. To create guardrails that would catch these errors before they cost me another weekend.

This article is the result of that rebuild.

Part 1 covered CLAUDE.md, agents, and MCP servers — the foundation.[1] This is Part 2: the automation layer. Skills, Hooks, and Commands. The systems that turn Claude Code from a powerful but unpredictable assistant into something closer to a disciplined teammate.

The architecture is simple:

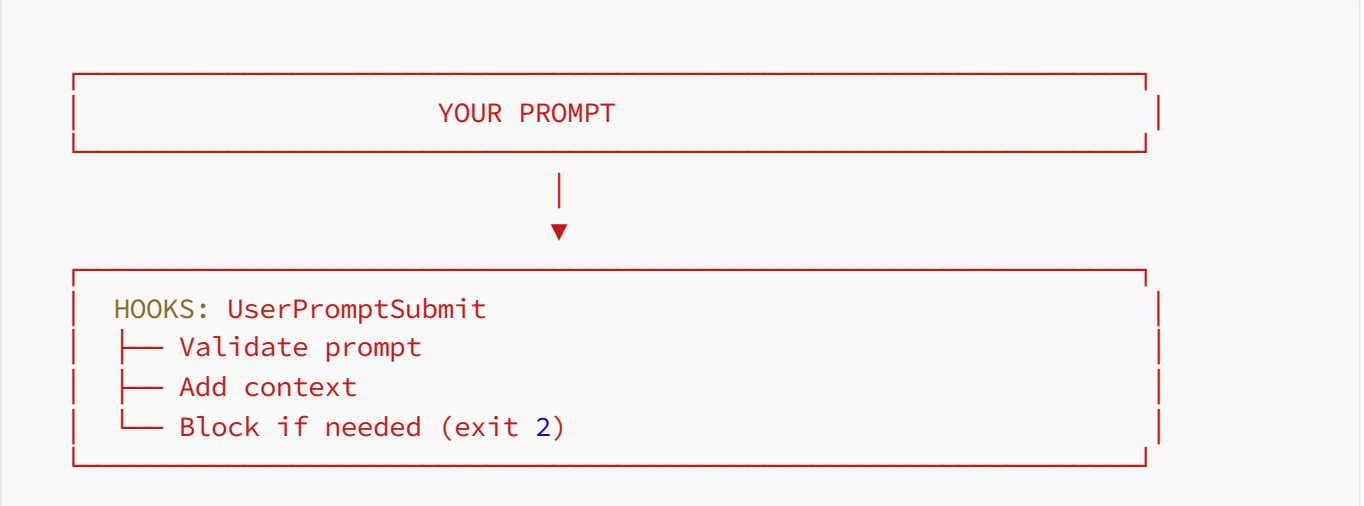
- **Skills** define what Claude *knows* (and force it to verify what it doesn't)
- **Hooks** control what Claude *can do* (deterministic guardrails)
- **Commands** automate what you *frequently ask* (semantic shortcuts)

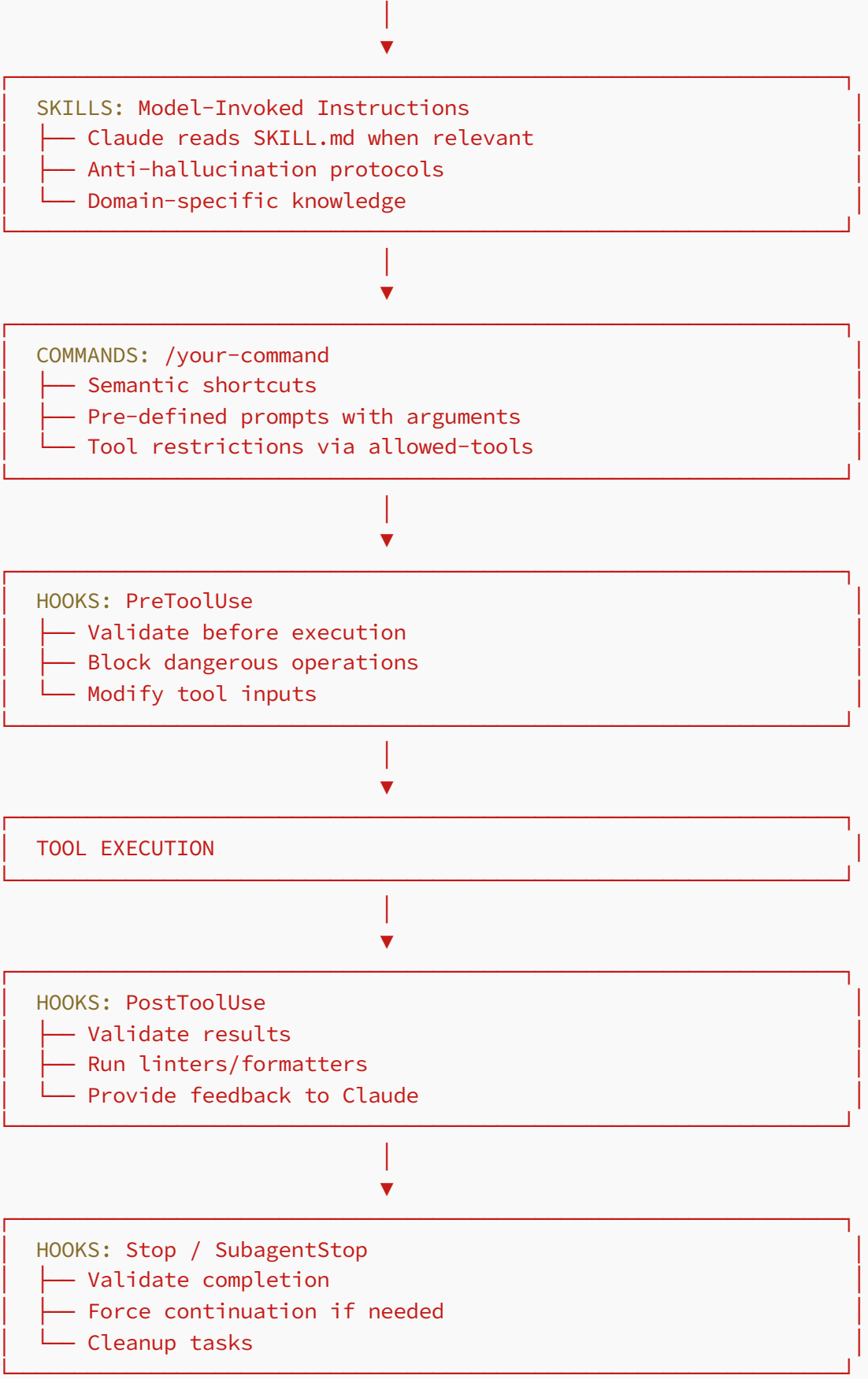
Let me show you how each piece works.

. . .

Part 1: The Automation Architecture

Before diving into components, let's understand how they fit together.





Pro tip: Run the `/context` command at the beginning of your session to ensure Claude Code properly routes requests to the right skills and agents. This simple habit prevents a lot of confusion.

The key insight: **Hooks are deterministic, Skills are semantic, Commands are shortcuts.** Each serves a different purpose:

Let's examine each in detail.

. . .

Part 2: Skills — The Anti-Hallucination Layer

Skills act as invisible shields — they verify before Claude can hallucinate.

Skills are folders containing a `SKILL.md` file that Claude reads when relevant to your task. Unlike slash commands (which you explicitly invoke), Skills are

model-invoked — Claude autonomously decides when to load them based on your request and the Skill's description.

The Structure

Skills live in two locations:

- **Personal:** `~/.claude/skills/skill-name/SKILL.md`
- **Project:** `.claude/skills/skill-name/SKILL.md`

Each Skill folder can contain supporting files: scripts, templates, data files. Claude reads the `SKILL.md` and has access to the entire folder.

The Frontmatter

The SKILL.md requires YAML frontmatter with specific fields:[3]

```
---
name: anti-hallucination
description: Use when working with APIs, libraries, or external services. Forces
allowed-tools: Read, Grep, WebSearch, WebFetch
---

# Anti-Hallucination Protocol
## MANDATORY VERIFICATION
Before generating ANY code involving external libraries or APIs:
1. **Search First**: Use WebSearch to find current documentation
2. **Verify Signatures**: Never assume method names or parameters
3. **Check Versions**: APIs change. Verify against the version in use.
4. **State Uncertainty**: If you cannot verify, say so explicitly.
## RED FLAGS
Stop and verify if you're about to write:
- API endpoints you haven't confirmed
- Method signatures from memory
- Configuration options without documentation
- Version-specific features without version check
## OUTPUT FORMAT
When presenting API/library code:
1. State the source of your information
2. Note the version verified against
3. Flag any assumptions made
```

Important fields:

The `description` field is critical. Claude uses it to decide when to load the Skill. A vague description means the Skill gets ignored. A specific description means it activates precisely when needed.

My Core Skills

Here's the Skill architecture I use daily:

1. Anti-Hallucination (shown above)

The most important Skill. Forces verification before generation. This single file has saved me more debugging hours than any other configuration.

2. Code Patterns

```
---
name: code-patterns
description: Use for API design, testing, Docker, CI/CD, or database work. Conta
---

# Code Patterns Reference
## API Design
- REST conventions: resource-based URLs, proper HTTP verbs
- Error handling: structured errors with codes
- Pagination: cursor-based for large datasets
## Testing Patterns
- Unit tests: isolate dependencies, test behavior not implementation
- Integration tests: real dependencies, cleanup after
- E2E: critical paths only, maintainable selectors
[... continues with Docker, CI/CD, database patterns ...]
```

3. Domain Knowledge

```
---
name: ml-domain
description: Use for machine learning, deep learning, or AI-related tasks. Conta
---

# ML Domain Knowledge
## Current SOTA (verify dates)
- Vision: Check papers from last 6 months
- NLP: Transformer variants evolving rapidly
- RL: PPO still baseline, but verify alternatives
## Common Pitfalls
- Data leakage in preprocessing
- Wrong metric for task type
- Overlooking class imbalance
[... domain-specific knowledge ...]
```

The Skill Invocation Flow

When you submit a prompt, Claude evaluates whether any Skill's `description` matches your request. If it does, Claude loads that Skill's content into context and follows its instructions.

```
User: "Help me integrate the Stripe API"

Claude's internal process:
1. Parse prompt → detect "API integration"
2. Check Skills → anti-hallucination matches
3. Load ~/.claude/skills/anti-hallucination/SKILL.md
4. Follow verification protocol before generating code
5. Search Stripe docs, verify endpoints, then generate
```

This is the power of Skills: they inject *protocols* into Claude's reasoning without you having to remember to ask for them.

Pre-Built Skills

Anthropic provides several official Skills for common tasks:[4]

You can install these or use them as templates for your own Skills.

. . .

Part 3: Hooks — Deterministic Event Handling

Hooks are the clockwork beneath the surface — deterministic, reliable, unstoppable.

If Skills are the semantic layer, Hooks are the deterministic layer. They fire on specific events and execute scripts or prompts with predictable behavior.

The 9 Hook Events

Claude Code supports 9 hook events — not 4, as some tutorials suggest:

Blocking means the hook can prevent the action from proceeding. **Matcher** means you can filter which tools/events trigger the hook.

The 3 Exit Codes

This is where many tutorials get it wrong. There are **3 exit codes**, not just “block” and “allow”:

The difference between `exit 1` and `exit 2` is crucial:

- `exit 1`: "Warning: this file is large" — Claude sees the warning but continues
- `exit 2`: "Blocked: cannot delete production files" — Action is prevented entirely

The 2 Hook Types

Hooks can be either **command** (bash scripts) or **prompt** (LLM decision-making):

Command hooks — Deterministic, fast, scripted:

```
{
  "type": "command",
  "command": "/path/to/validation-script.sh",
  "timeout": 60
}
```

Prompt hooks — Contextual, LLM-powered validation:

```
{
  "type": "prompt",
  "prompt": "Analyze this file write operation. If it modifies critical config f
  "timeout": 30
}
```

The official documentation actually **recommends prompt hooks for contextual validation**. They're slower but can make nuanced decisions that scripts cannot.

Timeout Defaults

- **Command hooks:** 60 seconds
- **Prompt hooks:** 30 seconds

You can override these per-hook with the `timeout` field.

Environment Variables

Hooks receive context through environment variables:

Hooks also receive JSON via stdin with full context about the event.

Configuration Structure

Hooks are configured in your settings files:

- User: `~/.claude/settings.json`
- Project: `.claude/settings.json`
- Local: `.claude/settings.local.json` (not committed)

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "bash ~/.claude/hooks/validate-write.sh",
            "timeout": 10
          }
        ]
      },
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "prompt",
            "prompt": "Analyze this bash command for safety. Block if it could d
          }
        ]
      }
    ],
    "PostToolUse": [
```

```
        {
          "matcher": "Write|Edit",
          "hooks": [
            {
              "type": "command",
              "command": "bash ~/.claude/hooks/post-write-lint.sh"
            }
          ]
        }
      ],
      "SessionStart": [
        {
          "matcher": "startup",
          "hooks": [
            {
              "type": "command",
              "command": "bash ~/.claude/hooks/session-init.sh"
            }
          ]
        }
      ]
    }
  }
```

Matcher Patterns

Matchers use regex-style patterns:

```
{
  "matcher": "Write",            // Exact match
  "matcher": "Edit|Write",       // Alternation
  "matcher": "Bash",             // Match Bash tool
  "matcher": "mcp__memory__.*",  // Match all memory MCP tools
  "matcher": "*"                 // Match all tools
}
```

Common tool names to match:

- `Task` — Subagent operations
- `Bash`, `Read`, `Write`, `Edit`, `Glob`, `Grep`
- `WebFetch`, `WebSearch`
- `mcp__<server>__<tool>` — MCP server tools

Practical Hook Examples

1. Block dangerous bash commands

```
#!/bin/bash
# ~/.claude/hooks/validate-bash.sh

# Read JSON from stdin
INPUT=$(cat)
COMMAND=$(echo "$INPUT" | jq -r '.tool_input.command // empty')
# Dangerous patterns
DANGEROUS_PATTERNS=(
  "rm -rf /"
  "rm -rf ~"
  "rm -rf \*"
  "> /dev/sda"
  "mkfs"
  "dd if="
  ":(){:|:&};:"
)
for pattern in "${DANGEROUS_PATTERNS[@]}"; do
  if [[ "$COMMAND" == *"$pattern"* ]]; then
    echo "BLOCKED: Dangerous command pattern detected: $pattern" >&2
    exit 2
  fi
done
# Check for production database access
if [[ "$COMMAND" == *"prod"* ]] && [[ "$COMMAND" == *"DROP"* || "$COMMAND" == *"
  echo "BLOCKED: Production database destructive operation" >&2
  exit 2
fi
exit 0
```

2. Auto-format after file writes

```
#!/bin/bash
# ~/.claude/hooks/post-write-lint.sh

for filepath in $CLAUDE_FILE_PATHS; do
  case "$filepath" in
    *.py)
      ruff format "$filepath" 2>/dev/null
      ruff check --fix "$filepath" 2>/dev/null
      ;;
    *.ts|*.tsx|*.js|*.jsx)
      npx prettier --write "$filepath" 2>/dev/null
      ;;
```

```
    *.go)
      gofmt -w "$filepath" 2>/dev/null
      ;;
  esac
done
exit 0
```

3. Validate file writes don't touch protected paths

```
#!/bin/bash
# ~/.claude/hooks/validate-write.sh

INPUT=$(cat)
FILE_PATH=$(echo "$INPUT" | jq -r '.tool_input.file_path // empty')
PROTECTED_PATHS=(
  "/etc"
  "/usr"
  "/bin"
  "/sbin"
  "$HOME/.ssh"
  "$HOME/.gnupg"
)
for protected in "${PROTECTED_PATHS[@]}"; do
  if [[ "$FILE_PATH" == "$protected"* ]]; then
    echo "BLOCKED: Cannot write to protected path: $protected" >&2
    exit 2
  fi
done
# Warn about config file modifications
if [[ "$FILE_PATH" == *".env"* ]] || [[ "$FILE_PATH" == *"config"* ]]; then
  echo "WARNING: Modifying configuration file" >&2
  exit 1  # Warn but allow
fi
exit 0
```

4. Session initialization

```
#!/bin/bash
# ~/.claude/hooks/session-init.sh

# Set up environment variables for the session
if [ -n "$CLAUDE_ENV_FILE" ]; then
  echo "export PROJECT_ROOT=$CLAUDE_PROJECT_DIR" >> "$CLAUDE_ENV_FILE"
  # Load project-specific env if it exists
```

```
  if [ -f "$CLAUDE_PROJECT_DIR/.env" ]; then
    while IFS= read -r line; do
      [[ "$line" =~ ^#.*$ ]] && continue
      [[ -z "$line" ]] && continue
      echo "export $line" >> "$CLAUDE_ENV_FILE"
    done < "$CLAUDE_PROJECT_DIR/.env"
  fi
fi
exit 0
```

JSON Output Control

For sophisticated control, hooks can return JSON to stdout:

```
{
  "continue": true,
  "suppressOutput": false,
  "systemMessage": "Warning: large file detected",
  "hookSpecificOutput": {
    "hookEventName": "PreToolUse",
    "permissionDecision": "allow",
    "permissionDecisionReason": "File size within limits",
    "updatedInput": {
      "file_path": "/modified/path.txt"
    }
  }
}
```

Key fields:

- `continue`: Whether Claude should proceed (false = stop entirely)
- `permissionDecision`: "allow", "deny", or "ask"
- `updatedInput`: Modify the tool's input parameters

. . .

Part 4: Commands — Semantic Shortcuts

One word to rule them all — Commands compress complexity into keystrokes.

Commands are the explicit automation layer. Unlike Skills (model-invoked) and Hooks (event-triggered), Commands are user-invoked with `/command-name`.

The Structure

Commands are Markdown files:

- **Project:** `.claude/commands/command-name.md`
- **Personal:** `~/.claude/commands/command-name.md`

The Frontmatter

Commands support powerful frontmatter options:

```
---
description: Review code for security vulnerabilities
model: opus
allowed-tools: Read, Grep, WebSearch
argument-hint: [file-path] [severity-level]
disable-model-invocation: false
---

Review the following file for security vulnerabilities: $1
Severity threshold: $2 (default: medium)
```

```
Focus on:
- SQL injection
- XSS vulnerabilities
- Authentication bypasses
- Sensitive data exposure
```

Frontmatter fields:

Note on argument-hint format: Use square brackets `[arg]` for arguments, not angle brackets `<arg>`. Both optional and required arguments use square brackets.

Arguments

Commands support two argument syntaxes:

Positional arguments (`$1`, `$2`, etc.):

```
---
argument-hint: [pr-number] [priority] [assignee]
---

Review PR #$1 with priority $2 and assign to $3.
```

All arguments (`$ARGUMENTS`):

```
---
argument-hint: [issue-description]
---

Fix the following issue: $ARGUMENTS
```

Dynamic Content

Commands can include dynamic content:

Bash execution with `!` prefix:

```
---
description: Commit staged changes
allowed-tools: Bash(git:*)
---

Current status:
!`git status`
Staged changes:
!`git diff --cached`
Create a commit message for these changes.
```

File references with `@` prefix:

```
Review the implementation:
@src/main.py

Compare with:
@tests/test_main.py
```

The `model` Field

You can force specific models for specific commands:

```
---
description: Complex architectural review
model: opus
---
```

Valid values: `opus`, `sonnet`, `haiku`, or full model IDs like `claude-3-5-haiku-20241022`.

Use Opus for complex reasoning tasks, Haiku for simple operations where speed matters.

The disable-model-invocation Field

This is powerful but often overlooked. When set to `true`, the command won't appear in the SlashCommand tool — meaning Claude cannot invoke it autonomously.

Use case: commands that should only run with explicit human approval.

```
---
description: Deploy to production
disable-model-invocation: true
allowed-tools: Bash(gh:*), Bash(kubectl:*)
---

Deploy current branch to production.
Checklist:
- [ ] All tests passing
- [ ] Security review complete
- [ ] Stakeholder approval received
Proceed with deployment.
```

Namespacing

Use subdirectories to organize commands:

```
.claude/commands/
├── git/
│   ├── commit.md      → /commit (project:git)
│   ├── pr.md          → /pr (project:git)
│   └── rebase.md      → /rebase (project:git)
├── deploy/
│   ├── staging.md     → /staging (project:deploy)
│   └── prod.md        → /prod (project:deploy)
└── review.md          → /review (project)
```

My Essential Commands

1. /debug — Systematic debugging

```
---
description: Debug an issue systematically
model: opus
allowed-tools: Read, Grep, Bash, WebSearch
argument-hint: [error-description]
---

## Debug Protocol
Error: $ARGUMENTS
### Step 1: Reproduce
Identify the exact steps to reproduce this issue.
### Step 2: Search
Before guessing, search for this error:
- Check official documentation
- Search GitHub issues
- Look for Stack Overflow solutions
### Step 3: Isolate
Find the minimal reproduction case.
### Step 4: Fix
Propose a fix with:
- The root cause
- The solution
- How to prevent recurrence
```

2. /review — Code review with specific focus

```
---
description: Review code for quality issues
allowed-tools: Read, Grep
```

```
argument-hint: [file-or-directory]
---

Review: $1
Focus areas:
1. **Logic errors**: Off-by-one, null checks, edge cases
2. **Security**: Input validation, injection, auth
3. **Performance**: N+1 queries, unnecessary loops
4. **Maintainability**: Naming, complexity, documentation
Format:
- 🔴 Critical: Must fix
- 🟡 Warning: Should fix
- 🔵 Suggestion: Consider
```

3. /commit — Smart commit message

```
---
description: Create a git commit with smart message
allowed-tools: Bash(git:*)
argument-hint: [optional-context]
---

Current changes:
!`git diff --cached --stat`
Detailed diff:
!`git diff --cached`
Create a commit message following conventional commits format.
Context: $ARGUMENTS
```

4. /test — Generate tests for a file

```
---
description: Generate tests for a file
allowed-tools: Read, Write, Bash
argument-hint: [file-path]
---

File to test:
@$1
Generate comprehensive tests covering:
- Happy path
- Edge cases
```

```
- Error conditions
Use the project's existing test framework and conventions.
```

. . .

Part 5: The Honest Downsides

Victory tastes sweeter when you've earned the right to trust your tools.

I've spent this article showing you powerful capabilities. Let me be honest about the limitations.

Skills Limitations

1. Discovery is imprecise

Claude's decision to load a Skill is based on semantic matching. Sometimes it loads Skills when it shouldn't. Sometimes it doesn't load them when it should.

Mitigation: Write very specific `description` fields. Test with various prompts.

2. Context consumption

Every loaded Skill consumes context window. Large Skills with many instructions can crowd out actual work.

Mitigation: Keep Skills focused. One concern per Skill. Use sub-files for reference data.

3. No guaranteed invocation

Unlike Hooks, Skills are *advisory*. Claude might acknowledge a Skill and then ignore parts of it.

Mitigation: For critical requirements, use Hooks (deterministic) instead of Skills (semantic).

Hooks Limitations

1. Stdin/stdout complexity

Parsing JSON in bash is painful. Complex hooks often need Python or Node wrappers.

Mitigation: Keep command hooks simple. Use prompt hooks for complex decisions.

2. Debugging is hard

When a hook fails silently or produces unexpected results, debugging is challenging. There's limited visibility into hook execution.

Mitigation: Log liberally. Test hooks in isolation before deploying.

3. Performance overhead

Every matched hook runs. Many hooks = slower tool execution. Prompt hooks are especially slow (they invoke the LLM).

Mitigation: Use specific matchers. Reserve prompt hooks for high-stakes decisions.

4. Parallel execution

All matching hooks run simultaneously. If you need sequential execution, you need a single hook that handles the sequence.

Commands Limitations

1. No conditional logic

Commands are templates, not programs. You can't do `if-else` based on context within the command file.

Mitigation: Create multiple commands for different scenarios, or let Claude's reasoning handle the conditionals.

2. Argument parsing is basic

No named arguments, no defaults, no validation. `$1` is either provided or empty.

Mitigation: Handle missing arguments in the prompt text (“If severity not provided, assume medium”).

3. Namespace collisions

Project commands override user commands. This can cause confusion if you forget which version is active.

Mitigation: Use `/help` to see active commands and their sources.

. . .

Part 6: Putting It All Together

Here’s how I combine everything into a coherent system.

The Complete Configuration

~/.claude/settings.json (User settings):

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "bash ~/.claude/hooks/validate-bash.sh",
            "timeout": 10
          }
        ]
      },
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "bash ~/.claude/hooks/validate-write.sh",
            "timeout": 5
          }
        ]
      }
    ],
```

```
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "bash ~/.claude/hooks/format-code.sh",
            "timeout": 30
          }
        ]
      }
    ],
    "SessionStart": [
      {
        "matcher": "startup",
        "hooks": [
          {
            "type": "command",
            "command": "bash ~/.claude/hooks/session-init.sh"
          }
        ]
      }
    ],
    "Stop": [
      {
        "hooks": [
          {
            "type": "prompt",
            "prompt": "Before finishing, verify: Did you follow the anti-halluci
          }
        ]
      }
    ]
  }
}
```

~/.claude/skills/ structure:

```
skills/
├── anti-hallucination/
│   └── SKILL.md
├── code-patterns/
│   ├── SKILL.md
│   ├── api-patterns.md
│   ├── testing-patterns.md
│   └── docker-patterns.md
├── debugging/
│   ├── SKILL.md
│   └── common-errors.md
└── domain-knowledge/
```

```
    ├── SKILL.md
    └── ml-reference.md
```

~/.claude/commands/ structure:

```
commands/
├── debug.md
├── review.md
├── commit.md
├── test.md
├── git/
│   ├── pr.md
│   ├── rebase.md
│   └── cleanup.md
└── deploy/
    ├── staging.md
    └── prod.md
```

The Workflow

1. **Start session:** SessionStart hook initializes environment
2. **Run** `/context`: Ensure proper routing to skills and agents
3. **Submit prompt:** UserPromptSubmit can add context or validate
4. **Skills load:** Claude loads relevant Skills based on task
5. **Execute command:** `/review src/` triggers review command
6. **PreToolUse:** Validates every tool call before execution
7. **PostToolUse:** Formats code, runs linters after writes
8. **Stop hook:** Verifies completion quality before finishing

Pro Tip: Sub-Agent Brainstorming

For complex tasks, try this approach: ask Claude Code to brainstorm with its relevant sub-agents. This collaborative method helps break down problems more effectively.

Key addition: Request that all recommendations be rigorously justified with sources or documentation. This ensures you get reliable, verifiable solutions rather than speculative answers.

Example prompt:

```
Brainstorm with the relevant sub-agents to design the authentication system
for this application. Everything must be rigorously justified with sources
or documentation.
```

This technique has significantly improved the quality of my AI-assisted development workflows. The sub-agents bring different perspectives, and the source requirement prevents hallucinated solutions.

. . .

Conclusion: The Paradox of Automated Trust

There's a paradox at the heart of this entire system.

I've built an elaborate architecture of Skills, Hooks, and Commands — all designed to constrain an AI, verify its outputs, and catch its mistakes. The goal is to make Claude more trustworthy by trusting it less.

But here's the thing: the prompt hooks? They use Claude to validate Claude. The Skills? They're instructions that Claude may or may not follow. The architecture assumes Claude is unreliable while simultaneously relying on Claude to implement the reliability.

This isn't a contradiction. It's a design pattern.

The human is still in the loop. The Hooks provide deterministic guardrails that don't depend on Claude's compliance. The Skills encode expertise that

Claude can leverage but could also ignore. The Commands provide shortcuts that are explicit rather than semantic.

The combination creates something more reliable than any single component.

What I've Learned

After months of iteration:

1. **Deterministic beats semantic for safety.** If something must not happen, use a Hook (exit 2), not a Skill.
2. **Semantic beats deterministic for capability.** Skills make Claude smarter. Hooks just make it safer.
3. **Explicit beats implicit for workflows.** Commands you invoke are more predictable than Skills that auto-load.
4. **Verification is not optional.** The anti-hallucination Skill has saved me more time than all other configurations combined.
5. **Start simple, iterate.** My current setup took months to develop. Start with one Hook, one Skill, one Command. Add complexity as you understand the system.

The Honest Assessment

Is this "war machine" metaphor accurate? Yes and no.

Yes: This configuration dramatically increases productivity and reduces errors. I can tackle larger projects with more confidence.

No: It's still an AI assistant. It still hallucinates. It still misses context. The architecture mitigates these issues; it doesn't eliminate them.

The goal was never to make Claude perfect. The goal was to make the failure modes visible, the guardrails explicit, and the verification systematic.

That weekend I lost to a hallucinated API signature? It hasn't happened since. Not because Claude stopped hallucinating — it hasn't. But because now, when it's about to hallucinate an API call, the anti-hallucination Skill kicks in. Claude searches the documentation. It verifies the signature. And if it can't verify, it says so.

That's not trust. That's verified reliability.

And verified reliability, it turns out, is far more useful than blind trust ever was.

. . .

✦ *DELANOE PIRARD* ✦
Artificial Intelligence Researcher & Engineer

🌐 *delanoe-pirard.com*
💻 *github.com/Aedelon*
💼 *linkedin.com/in/delanoe-pirard*
𝕏 *x.com/0xAedelon*

👉 This article did help you ? Clap + Follow for the next one.

. . .

Resources

Official Documentation

- Anthropic. "CLAUDE.md Configuration." Claude Code Documentation. https://code.claude.com/docs/en/claude-md
- Anthropic. "Agent Skills Overview." Claude Code Documentation. https://code.claude.com/docs/en/skills

- Anthropic. “SKILL.md Format.” Claude Code Documentation. https://code.claude.com/docs/en/skills
- Anthropic. “Pre-built Skills.” Claude Blog. https://claude.com/blog/skills
- Anthropic. “Hooks Reference.” Claude Code Documentation. https://code.claude.com/docs/en/hooks
- Anthropic. “How to Configure Hooks.” Claude Blog. https://claude.com/blog/how-to-configure-hooks
- Anthropic. “Slash Commands.” Claude Code Documentation. https://code.claude.com/docs/en/slash-commands

Additional Resources

- Disler. “Claude Code Hooks Mastery.” GitHub. https://github.com/disler/claude-code-hooks-mastery
- Shilkov, M. “Inside Claude Code Skills.” https://mikhail.io/2025/10/claude-code-skills/
- Anthropic. “Equipping Agents for the Real World.” https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills

Further Reading

- Vectara Hallucination Leaderboard — LLM accuracy benchmarks
- Anthropic Economic Index — Claude Code usage patterns

Claude Code | Ai Tools | Automation | Developer Productivity | Programming

Written by Delanoe Pirard

2.3K followers · 57 following

Follow

AI Researcher, AI Engineer. Building autonomous systems. Reinforcement Learning, Computer Vision. Brussels 🇧🇪 https://www.delanoe-pirard.com/

Responses (1)

robinmin he

What are your thoughts?

B *i* Cancel Respond

Sam Miller
Dec 28, 2025

Amazing article. Very well structured with helpful tips. Thanks

👏 Reply

More from Delanoe Pirard

Delanoe Pirard

Transformers Are Dead. Google Killed Them — Then Went Silent

A deep dive into Google's neural long-term memory architecture that claims 2M+ token context windows with O(n) complexity. The data is...

Dec 17, 2025 28

Delanoe Pirard

YOLO Is Dead. Meet RF-DETR, the Model That Just Crushed 10 Years of Computer Vision Dominance.

RF-DETR Shatters the 60 AP Barrier — And YOLO's Decade-Long Reign With It

Dec 7, 2025 18

In AI Advances by Delanoe Pirard

Why Yann LeCun Bet $3.5 Billion on World Models Over LLMs

After 12 years as Meta's Chief AI Scientist, the Turing Award winner walked away to prove Silicon Valley is betting on the wrong horse.

Dec 20, 2025 50

Delanoe Pirard

USC Just Built Artificial Neurons That Could Make GPT-5 Run on 20 Watts

After 70 years of von Neumann architecture, researchers create neurons that think like nature—using silver ions, not electrons. Here's...

Dec 23, 2025 21

See all from Delanoe Pirard

Recommended from Medium

In Dev Geniusby JP Caparas

Claude Code, but cheaper: GLM-4.7 on Z.ai with a tiny wrapper

Keen to try out Claude Code extensively but don't want to fork substantial money just yet? This guide is for you!

Dec 22, 2025 1

Marta Fernández García

Don't Use LLMs as OCR: Lessons Learned from Extracting Complex Documents

Recently, I had to work with complex documents—mostly PDFs—and send them to an LLM to answer questions about their content.

Dec 26, 2025 22

Marco Kotrotsos

How to Use Google Antigravity: A Complete Guide.

Agent-First Development

Dec 22, 2025

In Coding Nexusby Tattva Tarang

A New Agent Memory System Just Dropped—And It Finally Fixes What We've Been Getting Wrong

A new state-of-the-art agent memory system just dropped, and it has completely changed how I think about "memory" in AI agents.

Dec 20, 2025 5

Javokhirbek Parpikhodjaev

Conductor for Claude Code

Introducing Conductor CC: Bringing Structured, Context-Driven Development to Claude Code

Dec 28, 2025 2

In AI Advancesby Delanoe Pirard

A 170M Model Just Beat GPT-4. Google's TITANS Explains Why Size Doesn't Matter

The neuroscience of AI memory: how test-time learning and surprise-gated memory are rewriting the rules of deep learning.

Dec 31, 2025 12

See more recommendations