WIP: Harness Engineering with Claude Code

About This Book

Overview

Claude Code is powerful out of the box. Anthropic has also given us primitives to extend it: hooks for automation, commands and skills for workflows, MCP for external tools, and CLAUDE.md for persistent context.

This book documents patterns for combining these primitives into a harness that delivers consistent quality and coordinates parallel agents.

The core workflow:

1. Generate a maintainable, implementable plan
2. Audit the plan (up to 5 iterations until no issues remain)
3. Parse into tasks with dependencies, forming a DAG
4. Audit each task in parallel (up to 5 iterations)
5. Execute in parallel, using the DAG to find unblocked work

Goals

- Parallelized agent workflows for delivering features
- Consistent code quality through hook injection patterns
- Low-drag environment for a single developer

Non-Goals

- An autonomous framework; we empower engineers, using their expertise to keep the AI aligned
- VCS integration; we handle commits manually (the jj squash workflow is preferred by the author)
- Building multiple features in parallel; we use parallel agents on a single feature, as task dependencies allow

Prerequisites

Basic familiarity with Claude Code: context windows, compaction, agents. If terms like "token limit" or "subagent" are unfamiliar, start with the Anthropic documentation.

Part 0: Quick Start

Get running in 5 minutes. Understand why later.

Step 1: Install gg

```
# Build from source (requires Zig 0.14+)
cd guerilla-graph
zig build -Doptimize=ReleaseFast
cp zig-out/bin/gg ~/.local/bin/  # or anywhere in PATH
```

Step 2: Initialize Your Project

```
cd your-project
gg init                    # Creates .gg/tasks.db
echo ".gg/" >> .gitignore  # Don't commit the database
```

Step 3: Configure Hooks

Create `.claude/settings.json`:

```
{
  "hooks": {
    "SessionStart": [{
      "hooks": [
        { "type": "command", "command": "gg workflow" },
        { "type": "command", "command": "[ -d .jj ] && cat .claude/hooks/jj_prime.md 2>/dev/null" }
      ]
    }],
    "UserPromptSubmit": [{
      "hooks": [{ "type": "command", "command": "cat .claude/hooks/engineering_principles.md 2>/dev/null" }]
    }],
    "SubagentStart": [{
      "hooks": [{
        "type": "command",
        "command": "gg workflow 2>/dev/null; cat .claude/hooks/engineering_principles.md 2>/dev/null"
      }]
    }]
  }
}
```

The `jj_prime.md` hook loads Jujutsu workflow context (squash workflow, bookmark management) only if the project uses jj (detected by `.jj/` directory).

Step 4: Add Engineering Principles

Create `.claude/hooks/engineering_principles.md` with your coding standards. Example for a Java/Gradle project:

```
# Engineering Principles

ultrathink

You are a senior software architect who cares deeply about maintainability.

## Practices

### code_exploration_and_planning
1. Before changes, explore the codebase systematically
2. Read the specific code sections you'll be modifying
3. Look for existing patterns and utilities
4. Never defer research. Get it done

### coding
1. Audit documentation of files you update (keep concise)
2. Audit tests of files you update (keep clean)
3. Apply Single Responsibility Principle
4. Apply DRY principle; look for existing abstractions
5. Watch for N+1 query issues
6. No backwards compatibility unless explicitly asked

### tools
1. Use `:compileJava` rather than `:build` unless running tests
```

```
2. After `./gradlew test`, read `build/reports/tests/test/index.html`
3. Use agents for exploration (>3 files), work directly for focused edits
```

The key: include **project-specific commands** and **tool guidance**, not just abstract principles.

Step 5: Copy Slash Commands

The guerilla-graph repo (where you built `gg`) also contains the slash commands. Copy them to your project:

```
cp -r /path/to/guerilla-graph/.claude/commands your-project/.claude/
```

The `_prompts/` and `_shared/` subdirectories contain reusable modules for the slash commands. You don't need to customize these; they work out of the box.

Step 6: Review CLAUDE.md

If you haven't already, run `/init` in Claude Code to create `CLAUDE.md` with your project's build commands, architecture, and patterns. Review the generated file and refine as needed. Claude learns your codebase automatically, but domain-specific context (business rules, team conventions) may need manual additions.

Step 7: Run Your First Feature

```
# Start Claude Code in your project
claude

# Generate a plan
/gg-plan-gen "Add user authentication with JWT tokens"

# Audit the plan (iterates until quality passes)
/gg-plan-audit

# Generate tasks with dependencies
/gg-task-gen

# Audit tasks (parallel agents verify each task)
/gg-task-audit auth

# Execute with parallel agents
/gg-execute auth
```

What Just Happened?

1. **gg workflow** injected on session start, so Claude knows how to use the task system
2. **engineering_principles.md** injected on every prompt for consistent quality
3. **/gg-plan-gen** explored your codebase and created PLAN.md
4. **/gg-plan-audit** iteratively refined until no critical issues
5. **/gg-task-gen** converted the plan to gg tasks with dependencies
6. **/gg-task-audit** verified each task has full implementation context
7. **/gg-execute** ran parallel agents, compiled after each wave, closed tasks

Quick Reference

```
gg workflow             # Full protocol (run after compaction)
gg ready                # Find unblocked work
gg start <task-id>      # Claim a task
```

```
gg complete <task-id>    # Mark done
gg task ls --plan <slug> # See all tasks in a plan
```

Now read on to understand *why* this works.

Part 1: The Problem

Where Vanilla Claude Code Falls Short

Claude Code's built-in task tracking (TodoWrite) works well for sequential work. But when we want to parallelize by running multiple agents on independent tasks, we need more structure.

The key insight: tasks have dependencies. Task B might require Task A's output. Tasks C and D might both modify the same file. Without tracking these relationships, parallel execution is guesswork.

A DAG (directed acyclic graph) solves this. Tasks become nodes, dependencies become edges. We query for "ready" tasks, those with all blockers complete, and spawn that many agents safely.

The tool we're missing is a task tracker that knows about dependencies. This could be as simple as bash and sqlite. For this workflow, we use gg (Guerilla Graph), a lightweight task tracker built specifically for this purpose.

Part 2: The Harness

Engineering Principles Injection

The core pattern: inject our standards into every prompt via hooks.

```
// .claude/settings.json
{
  "hooks": {
    "SessionStart": [{
      "hooks": [{ "type": "command", "command": "gg workflow" }]
    }],
    "UserPromptSubmit": [{
      "hooks": [{ "type": "command", "command": "cat .claude/hooks/engineering_principles.md 2>/dev/null" }]
    }],
    "SubagentStart": [{
      "hooks": [{
        "type": "command",
        "command": "gg workflow 2>/dev/null; cat .claude/hooks/engineering_principles.md 2>/dev/null"
      }]
    }]
  }
}
```

Now every response, including agents, sees the same engineering principles. No drift.

What goes in engineering_principles.md: - Coding standards (assertions, function length limits, explicit types) - Naming conventions - What to avoid (over-engineering, backwards compatibility unless asked) - Tool usage patterns

Commands as Reusable Workflows

Slash commands encode multi-step processes that would otherwise require lengthy prompts.

```
.claude/commands/
  gg-plan-gen.md       # Generate plan from feature spec
  gg-plan-audit.md     # Audit plan for quality issues
  gg-task-gen.md       # Generate tasks from plan
  gg-task-audit.md     # Audit tasks for implementability
  gg-execute.md        # Parallel agent execution
```

Each command is a markdown file with YAML frontmatter for arguments:

```
---
description: Generate plan from feature spec
args:
  - name: feature
    description: Feature description or path to spec file
    required: true
---

[Detailed instructions for Claude...]
```

When commands beat natural language: - Multi-step processes you repeat - Workflows that need consistency across uses - Instructions too long to type each time

MCP for Domain Verification

MCP servers provide tools beyond Claude's built-ins. The pattern: use them for domain-specific verification.

Example: `zig-docs` MCP server for Zig standard library lookup. Instead of Claude guessing at APIs, it can verify:

```
Use the zig-docs MCP server to verify ArrayList API before using it.
```

This catches errors that would otherwise require a compile cycle to find.

Context Recovery

SessionStart hooks recover context after compaction or new sessions:

```
"SessionStart": [{
  "hooks": [{ "type": "command", "command": "gg workflow" }]
}]
```

The `gg workflow` command outputs the full workflow protocol, a ~100 line reference covering task ID format, core commands, parallel execution patterns, and DAG rules. Sample excerpt:

```
EXECUTION PHASE (AI Agent Workflow)
  Find work, claim tasks, execute, and complete:

  - Find work: gg ready <plan-slug> --json
  - Spawn agents: N agents for N ready tasks (maximize parallelism)
  - Claim and read: gg start <slug:NNN> --json
  - Execute work
  - Complete: gg complete <slug:NNN>
  - Repeat until feature complete

CORE RULES:
  - Track ALL work in gg (no TodoWrite tool, no markdown TODOs)
  - Use 'gg ready' to find work (never guess task IDs)
  - Ready task count = parallelism capacity (5 ready = 5 agents)
```

Claude reads this on every session start and knows how to proceed.

Part 3: Hard-Won Lessons

Words That Work

Certain words trigger careful behavior:

Word	Effect
"maintainability"	Triggers careful design, avoids clever tricks
"implementability"	Forces concrete thinking, verifiable steps
"DRY"	Prevents duplication, looks for existing patterns
"YAGNI"	Stops over-engineering, feature creep

Words That Don't

Word	Problem
"robust"	Vague, leads to defensive bloat
"comprehensive"	Invites scope creep
"flexible"	Abstraction bait, premature generalization

Thinking Modes

Claude has extended thinking modes that allocate more compute to reasoning before responding. The keyword "ultrathink" triggers the deepest reasoning mode, and Claude will think longer and more carefully before acting.

Put "ultrathink" at the top of your engineering_principles.md:

```
# Engineering Principles

ultrathink

You are a senior software architect...
```

Now every prompt triggers deep reasoning. Yes, it costs more tokens. Yes, it's worth it. The difference in code quality, especially for multi-step planning and architectural decisions, is significant.

When to use ultrathink: Always. The cost is negligible compared to fixing bad code.

Audit Convergence

Iterative refinement with a stopping condition:

1. Generate artifact (plan, task, code)
2. Audit for issues (categorize: Critical, High, Medium, Low)
3. If Critical or High issues exist, fix and re-audit
4. Max 5 iterations, then stop and report

The key insight: **set a max iteration limit**. Should the audit not converge, the engineer must make a judgement call. Another round of audits, or to move forward.

Course Correction Patterns

- actively review each edit
- realign the ai if it strays
- worst case `jj restore`

Checkpointing good states:

- always start your feature with a fresh VCS state
- if things go wrong, you can restore

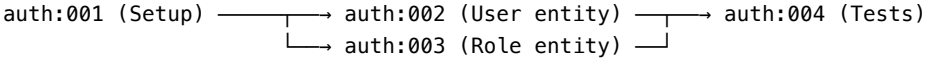
Part 4: Parallel Agent Coordination

Why TodoWrite Fails at Scale

- **Single-threaded mental model**: One task at a time, no parallelism
- **No dependency awareness**: Can't express "B requires A"
- **Lost on compaction**: Todo state exists only in context

DAG-Based Task Management

Tasks as nodes, dependencies as edges. A task is "ready" when all its blockers are complete.



After completing auth:001, both auth:002 and auth:003 are ready. They can run in parallel. auth:004 waits for both.

Ready task count = parallelism level. Five ready tasks means you can spawn five agents.

The GG Workflow

```
1. /gg-plan-gen <feature>     → PLAN.md with phases, architecture
2. /gg-plan-audit             → Iterate until no Critical/High issues
3. /gg-task-gen               → Tasks in gg with dependencies set
4. /gg-task-audit <plan>      → Iterate until all tasks are implementable
5. /gg-execute <plan>         → Parallel agents, compilation gates
```

Each step has quality gates. No skipping audits; they prevent rework later.

Wave Execution

```
Wave 1: Find ready tasks → Spawn N agents → Wait for all → Compile → Close tasks
Wave 2: Find newly ready tasks → Spawn N agents → ...
Repeat until all tasks complete
```

Critical rules: - One task per agent - Compilation gate before closing ANY task in the wave - If build fails, keep all tasks open, fix, retry - File-level serialization: tasks touching same file must be chained

Recovery from Failed States/Compaction

```
# See what's stuck
gg task ls --status in_progress --plan <plan>

# Reset and retry
gg update <task-id> --status open
```

```
jj restore <files>  # or: git checkout <files>
/gg-execute <plan>
```

Part 5: Reference

Complete Harness Structure

```
.claude/
  settings.json              # Hooks, permissions, model defaults
  settings.local.json        # User overrides (gitignored)
  hooks/
    engineering_principles.md    # Injected on every prompt
    jj_prime.md                  # Jujutsu workflow (SessionStart, if .jj/ exists)
  commands/
    README.md                # Documentation for slash commands
    gg-plan-gen.md
    gg-plan-audit.md
    gg-task-gen.md
    gg-task-audit.md
    gg-execute.md
    _prompts/                # Agent prompt templates
      explore-codebase.md
      implement-task.md
      audit-task.md
      audit-plan.md
      review-work.md
    _shared/                 # Reusable modules
      project-context.md
      dependency-analysis.md
      quality-criteria.md
      quality-standards.md
      code-verification.md
      file-verification.md
      control-flow.md
      fix-patterns.md
      json-parsing.md
      planning-checklist.md
      task-template.md
CLAUDE.md                    # Domain memory, architecture, patterns
.gg/
  tasks.db                   # SQLite: plans, tasks, dependencies
```

Quick Patterns

Situation	Pattern
Quality drifting	Add to engineering_principles.md
Repeated workflow	Create a command
Need domain verification	Add MCP server
Context lost	Add SessionStart hook
Parallel work needed	Use gg with dependencies
Audit looping	Set max iterations

Addendum

Resources

- Anthropic Documentation
- Claude Code GitHub

Acknowledgments

- Indie Dev Dan; practical workflow patterns
- The Claude Code team at Anthropic