Member-only story

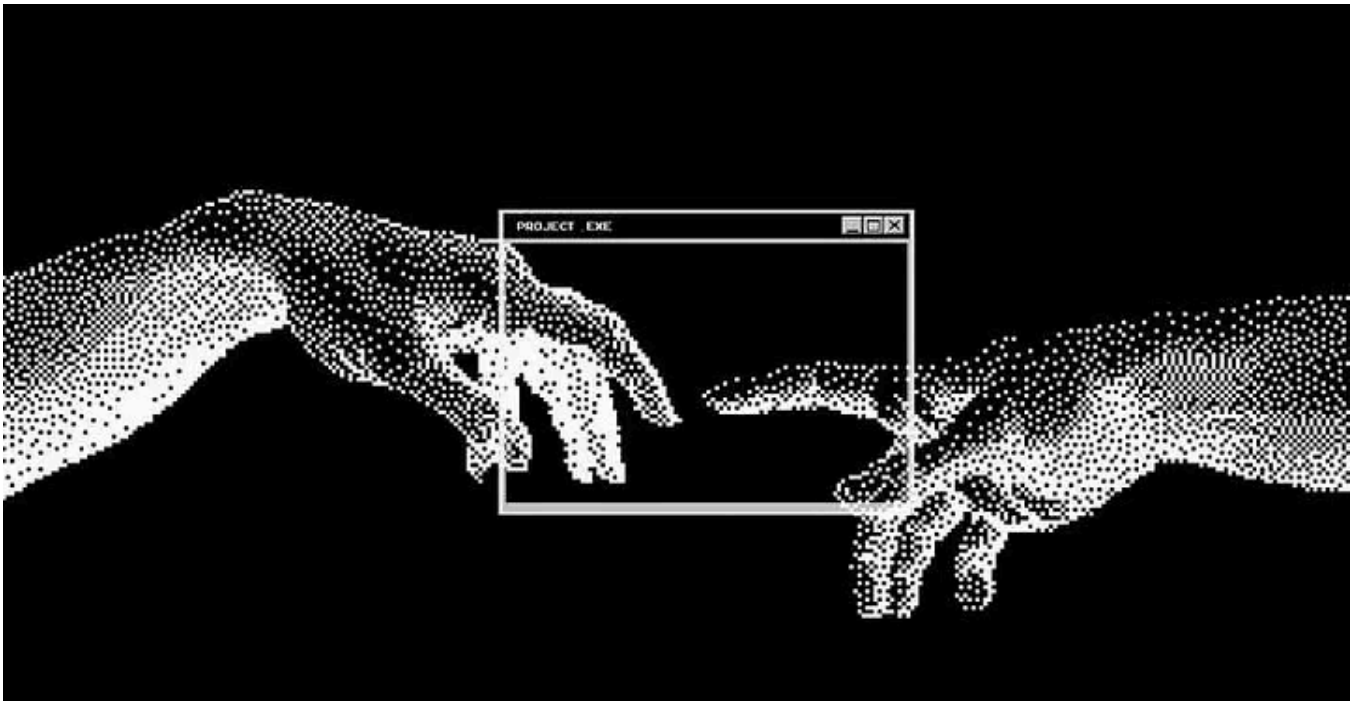
I mastered the Claude Code workflow

11 min read · 6 days ago

Ashley Ha Follow

Listen Share More

You 🤝 Claude Code



the TL;DR

Never exceed 60% context. Split work into 4 phases: Research → Plan → Implement → Validate. Clear context between each. Save everything to a sym-linked thoughts/

directory (use npx humanlayer thoughts init).

> At the middle & end I provide a free download all .claude commands & agents templates I talk about here and I'll explain how you can use them effectively for your codebase.

One of the most powerful points I want to make, is about the sym-linked 'thoughts' directory here. The important thing to note is that this directory can be shared globally, accross repos, so everyone on your team + org have access to the documents that claude is editing + building from. It's extremely powerful.

Now onto the good stuff.

Context heaven

Six months ago, my conversations with AI coding assistants looked something like this: I'd start working on a feature, load up a bunch of files, ask questions, get responses, make changes, hit errors, add more files for context, ask more questions. before I knew it, I was at 95% context capacity with half-implemented code and a dozen *"final_implementation_final_feature.md"* files💀

Shifting my focus on context engineering changed that. *(huge s/o to the humanlayer team, specifically Dex who explains this incredibly well, who I was heavily inspired by.)* I can now ship thousands of lines of code with more confidence using a structured workflow that keeps my context clean, plans solid, and implementation outcomes **a lot more predictable.**

Context Engineering Is All You Need

It's really not about having more context, it's about having the right context at the right time.

The more you allocate into the context window, the worse the outcomes (both quality & unexpected behavior) and I've found this to be true regardless of which LLM you're using.

My personal rule: **never let context exceed 60%.** I'm always checking with `/context` during my sessions.

For reference, this is my context going into a new conversation

So, roughly ~32% at the start of any convo.

I typically break my workflow into ***four core phases***, clearing context between each. For example it might look like:

1. Research: My claude prompt: **"/research_codebase + what feature I want to implement next or general codebase research" → /context and /clear context if the research was over 60% which typically is and so on for the remaining 3 phases.**

2. Plan: My claude prompt: **"/create_plan + @thoughts/research/researched_feature.md" → /context → 70% → /clear**

3. Implement: My claude prompt: **"/implement_plan + @thoughts/plans/research_and_planned_feature.md Phase 1 only of this plan" → /context → 67% → /clear**

4. Validate: My claude prompt: **"/validate_plan + @thoughts/plans/research_and_planned_feature.md + I implemented phase 1 of this plan doc, please validate it" → /context → 42% → continue with thread (i.e dont clear context)**

Each phase has a specific job. Each phase produces artifacts that the next phase consumes.

Yeah, it's... a lot so let me break it down

My Claude Code Workflow

Free download of all the .claude commands & agents

Phase 1: Research (@research_codebase.md)

Goal: **Understand what exists & what needs to change. NO code should be written during this phase.**

When I start a new feature, I never dive into implementation. I **always** research first, invoking my custom claude command (.claude/commands/research_codebase.md):

```
/research_codebase + {What I want to research or build or feature I want to imp
```

Then after my command, I provide my research question. For example, when I needed to understand browser page management in a web automation framework:

> /research_codebase + How does the current browser page management system work? or

> /research_codebase + Where are pages created, tracked, and cleaned up?

The research command spawns parallel sub-agents to investigate different aspects simultaneously:

- A **codebase-locator agent** finds all relevant files
- A **codebase-analyzer agent** examines how the current implementation works
- A **thoughts-locator agent** searches for any existing documentation or past research
- A **thoughts-analyzer agent** extracts insights from relevant documents

These agents work in parallel, returning specific file paths and line numbers.

The research phase finishes by generating a comprehensive research document in my thoughts/ directory:

> `thoughts/shared/research/2025-10-09_browser-page-management.md`

This document is comprehensive and includes:

- Summary of findings
- Code references with file:line numbers *very important*
- Architecture insights
- Open questions that need clarification

By spawning parallel agents, I get comprehensive answers fast without manually reading dozens of files. And because everything gets saved to thoughts/ directory, myself, my team and claude can reference it later without keeping it in context.

Phase 2: Plan (@create_plan.md)

Goal: Create a detailed, iterative implementation plan.

After research, I clear my context and start fresh with planning:

```
@create_plan.md thoughts/shared/research/2025-10-09_browser-page-management.md
```

This command reads my research document and begins an interactive planning session. After claudes initial response, I typically iterate the plan roughly 5 times before finalizing it reading the .md thoroughly.

> This plan doc is our source of truth

also why iterate so much? Because the first draft is never complete. The planning phase is interactive:

1. First draft: Claude reads the research, asks clarifying questions

2. Second draft: I provide answers, Claude refines the approach

3. Third draft: We discuss design tradeoffs, Claude updates phases

4. Fourth draft: We identify edge cases, Claude adds success criteria

5. Fifth draft: Final review, everything is actionable

Each iteration makes the plan more concrete. By the time I'm done, the plan has:

- Clear phases with specific changes
- Exact file paths to modify
- Code snippets showing what to add/change
- Automated verification (tests, linting, type checking)
- Manual verification (things humans need to test)

- Success criteria for each phase

The final plan gets saved to:

```
thoughts/shared/plans/2025-10-09-browser-page-management.md
```

Here's an example of what a phase in our plan might looks like:

```
Phase 1: Add Page Tracking to BrowserContext
```

```
Overview
Add a page registry to BrowserContext to track all created pages
and their lifecycle state.
Changes Required:
1. Update BrowserContext class
File: src/browser/browser_context.py
class BrowserContext:
    def __init__(self, ...):
        self._pages: Dict[str, Page] = {}
        self._active_page: Optional[Page] = None
Success Criteria:
Automated Verification:
- [ ] Unit tests pass: uv run pytest src/browser/tests/
- [ ] No linting errors: make lint
- [ ] Type checking clean: make check
Manual Verification:
- [ ] Can track multiple pages simultaneously
- [ ] Page cleanup works in headed browser mode
- [ ] No memory leaks with repeated page creation
```

The key insight here is that spending time iterating on the plan saves **hours** during implementation and bad code. A solid plan means I rarely get stuck or need to backtrack.

The research command already found the relevant lines of code so our planning command got saved a ton of context it can now use to implement the plan more effectively.

Humaine Interface by Ashley Ha is a reader-supported publication. To receive new posts and support my work, consider becoming a free or paid subscriber.

Phase 3: Implement (@implement_plan.md)

Goal: Execute one phase at a time with confidence.

After planning, I clear my context again and start implementing:

```
@implement_plan.md thoughts/shared/plans/2025-10-09-browser-page-management.md
```

My rule: implement one phase at a time.

The implementation command:

1. Reads the plan completely

2. Reads all files mentioned in Phase 1

3. Makes the changes specified

4. Runs automated verification

5. Pauses for manual testing

That last point is crucial. After passing all automated checks, Claude tells me:

```
Phase 1 Complete - Ready for Manual Verification
```

```
Automated verification passed:
- ✓ Unit tests pass
- ✓ No linting errors
- ✓ Type checking clean
Please perform manual verification:
- [ ] Can track multiple pages simultaneously
- [ ] Page cleanup works in headed browser mode
Let me know when manual testing is complete.
```

I then manually test the feature. If it works, I tell Claude to proceed to Phase 2 (If my context is still less than 60%). If not, I report what failed and we fix it before moving forward.

This approach prevents a common trap: implementing everything, then discovering Phase 1 had a critical bug that breaks everything after it.

> Key insight: One phase at a time = continuous validation = fewer surprises

Phase 4: Validate (@validate_plan.md)

Goal: Systematically verify the entire implementation.

After implementing all phases (or even just one), I clear context one more time and validate:

```
@validate_plan.md thoughts/shared/plans/2025-10-09-browser-page-management.md
```

The validation command:

1. Reads the plan

2. Checks recent git commits

3. Runs all automated verification commands

4. Reviews code changes against plan specifications

5. Generates a comprehensive validation report

The report shows:

- ✓ What was implemented correctly

- ⚠️ Any deviations from the plan

- ✗ Issues that need fixing

- Manual testing checklist

This final step catches things like:

- “Phase 2 added extra validation (good!)”

- “Missing error handling for edge case X (needs fix)”

- “Automated tests pass but manual testing reveals UX issue”

Key insight: Validation is your safety net. It ensures you didn’t miss anything before shipping.

The Secret Weapon: The thoughts/ Directory

Throughout this workflow, everything gets saved to my thoughts/ directory:

```
thoughts/
```

```
├── ashley/
│   ├── tickets/
│   │   └── eng_1478.md          # Original linear ticket
```

```
│   └── notes/
│       └── notes.md              # Personal observations and brain
dumps
├── shared/
│   ├── research/
│   │   └── 2025-10-09_browser-page-management.md
│   ├── plans/
│   │   └── 2025-10-09-browser-page-management.md
│   └── prs/
│       └── pr_456_browser_pages.md
└── searchable/                    # Symlinked for searching
```

Why is this powerful?

1. Persistent memory: Research and plans survive context clearing

2. Reusable knowledge: Future features can reference past decisions

3. Team collaboration: Shared research helps everyone

4. Audit trail: I can see why decisions were made months later

> The searchable/ directory contains hard links to all documents, making it easy for Claude's search tools to find relevant context when needed.

Why This Workflow Works

This workflow works well because it aligns with how both humans and AI work best:

For AI (Claude):

- Clear objectives: Each phase has one job
- Relevant context: Only what's needed, nothing more
- Verification loops: Automated checks prevent drift
- Memory in files: No need to "remember" across sessions

For humans (me & you):

- Cognitive load: One phase at a time is manageable

- Confidence: Solid plans reduce uncertainty
- Quality: Multiple validation points catch issues early
- Speed: Parallel research and clear phases = faster delivery

For the codebase:

- Documentation: Every feature has research + plan
- Consistency: Following patterns discovered in research
- Maintainability: Future developers understand decisions
- Quality: Systematic validation = fewer bugs

Tips

1. Don't skip research

Even if you think you know the codebase, run research. You'll discover patterns and edge cases you forgot about.

2. Iterate the plan more than you think

My first plans were always too shallow. Now I budget 30–45 minutes for planning and iterate 5+ times. The time investment pays off.

3. Be ruthless about context

If context hits 60%, stop and ask yourself: "What can I save to a file and reference later?" Usually, it's research findings or plan details.

4. Manual testing matters

Automated tests catch code-level issues. Manual testing catches UX issues, performance problems, and real-world edge cases. Do both.

5. Update plans during implementation

If you discover something new during Phase 2 that affects Phase 3, update the plan file. Keep it as the source of truth.

6. **Use the thoughts directory religiously**

Every research document, plan, and note goes in thoughts/.

Future you will thank present you :)

Want to try this workflow? Here's how to begin:

1. Set up your command files (or download the templates I provide here)

Create .claude/commands/ (or .cursor/commands/) directory with:

- research_codebase.md
- create_plan.md
- implement_plan.md
- validate_plan.md

You can adapt the commands I've shared or create your own variations.

2. Create your sym-linked thoughts directory

```
mkdir -p thoughts/{personal,shared,searchable}
```

```
mkdir -p thoughts/shared/{research,plans,prs}
```

```
mkdir -p thoughts/personal/{tickets,notes}
```

3. Try one feature with the full workflow

Pick a small feature. Don't try to learn everything at once:

1. Research → clear context

2. Plan (iterate!!!) → clear context

3. Implement (one phase) → clear context

4. Validate → clear context

Repeat~

4. Observe what works

After your first feature, reflect:

- Where did you get stuck?
- What could be clearer in your commands?
- Did you maintain <60% context?

Iterate on the workflow itself.

In Conclusion, The Meta-Skill: Context Engineering

Here's what I've really learned: working with AI isn't always about asking the right questions, it's about engineering the right context.

Every time you invoke Claude, you're loading a specific context into its working memory. Just like you wouldn't compile your entire codebase to change one function, you shouldn't load your entire project history to implement one feature.

The workflow I've shared is context engineering in practice:

- Research: Load context to understand
- Plan: Load context to design
- Implement: Load context to execute
- Validate: Load context to verify

Each phase has a focused purpose, produces artifacts & clears for the next.

If you're still struggling with coding alongside AI and frustrated hitting context limits, try this workflow.

Start small. Iterate. Make it your own.

Thanks for reading,

— Ashley Brooke Ha

. . .

Leave a comment

Questions? Want to share your own workflow? I'd love to hear from you. You can find me at ashleyha0317@gmail.com or on X @ ashleybcha.

. . .

Appendix: Quick Reference

The Four-Phase Workflow

1. Research (@research_codebase.md)

- Understand existing code
- Spawn parallel agents
- Generate research document
- Clear context

2. Plan (@create_plan.md)

- Create implementation plan
- Iterate 5+ times
- Define success criteria
- Clear context

3. Implement (@implement_plan.md)

- Execute one phase at a time
- Run automated verification
- Perform manual testing
- Clear context between phases

4. Validate the plan that was implemented (@validate_plan.md)

- Systematic verification
- Generate validation report
- Ensure nothing missed
- Clear context

Context Management Rules

- Never exceed 60% context capacity
- Clear context after each major phase
- Save everything to thoughts/ directory (npx humanlayer thoughts init)
- Reference files instead of keeping in memory
- Use parallel agents to gather information efficiently

Directory Structure

```
thoughts/
```

```
├── [personal]/           # Your personal notes/tickets
│   ├── tickets/
│   └── notes/
├── shared/               # Team-shared documents
│   ├── research/
│   ├── plans/
│   └── prs/
└── searchable/          # Hard links for searching
```

. . .

Thanks for reading Humaine Interface! This post is public and free so feel free to share it as it helps grow my publication.

Share

References & Notes

Notes

In addition to the 4 core phases I described, I also include in this zip folder other helpful commands such as a linear.md, commit.md, founder_mode.md, and describe_pr.md, all of which i've found have really helped make the process of coding with an agent really flawless beginning to end. From initial codebase + feature research, to plan implementation, to commiting to github, examining relavent pull requests, creating linear tickets and more. I really recommend checking out the videos and resources below as well as Dex and the team at humanlayer have done a fantastic job in really pioneering these commands and agents. so big THANKS AGAIN to their team❤

Videos

- **Advanced context engineering for coding agents**
- **12-Factor Agents: Patterns of reliable LLM applications — Dex Horthy, HumanLayer**
- **Advanced Context Engineering for Agents**

Blog Posts & Articles

- The New Skill in AI is Not Prompting, It's Context Engineering — Phil Schmid on Context Engineering
- Effective context engineering for AI agents — Anthropic on Context Engineering for AI Agents

Tools & Resources

- Humanlayer GitHub — Context management and thoughts directory tooling
- Let's Connect on X

One final note

I've started developing my very first open source library "Claude Code Workflow (ccw)" that will enable developers to easily create .claude Commands & Agents tailored to their specific codebase. If this sounds like something that interests you, email me at ashleyha0317@gmail.com

Artificial Intelligence | Claude | AI | Anthropic Claude | Technology

Follow

Written by Ashley Ha

377 followers · 64 following

Author & Creator of Humaine Interface | Data Science & ML @ UC Berkeley | Prev Data Scientist @ Rivian | AI & Technology

Responses (2)

Bgerby

What are your thoughts?

mohamad shahkhajeh
3 days ago

Have you tested how performance or merge conflicts behave when multiple repos access the sym-linked thoughts directory concurrently?

 --  1 reply Reply

Riley Gerszewski
5 hours ago

This is brilliant. Thank you for sharing.

-- Reply

More from Ashley Ha

Ashley Ha

Let’s learn about Universal Language Model Fine-tuning, ULMFiT

Introduction to ULMFiT

Dec 29, 2022

Ashley Ha

Unit Testing for Data Science — with a Linear Regression Example

The following are the topics I will cover in this blog post:

Jan 5, 2023

Ashley Ha

Multinomial Classification with Unbalanced data using Random Forest — Machine Learning with...

Multinomial classification problems are unique, in that, instead of classifying x-number of features into class “A” or class “B”, we are...

Dec 22, 2022 •••

Ashley Ha

Data Science Fundamentals — Hypothesis Testing

Mastering the Art of Hypothesis Testing in Data Science

May 25, 2023 •••

See all from Ashley Ha

Recommended from Medium

In MeetCyber by Jules May

Vibe coding: the antidote

LLMs have shown themselves to be a terrible way to write programs. But the problem they address is real, and we aren't solving it any other...

Oct 16

Pawel

Claude Skills: The AI Feature That Actually Solves a Real Problem

Yesterday, Anthropic quietly released what might be the most practical AI feature of 2025. It's not flashier models or better benchmarks...

6d ago

Jan Kammerath

A New Era Of AI App Development: Apple Cracked LLM & AI Integration

The new iOS 26 and Apple Intelligence sparked a lot of debate around Apple's approach to AI, ML and LLMs specifically. A lot of that debate...

6d ago

In Coding Beauty by Tari Ibaba

Google Just Made Gemini CLI Even More Insane 🤯

Massive new upgrade to this powerful CLI

5d ago

In nginity by Reza Rezvani

Claude AI and Claude Code Skills: Teaching AI to Think Like Your Best Engineer

4d ago

In The Context Layer by Jannis

Claude’s New “Skills” Show How Anthropic Is Layering Intelligence on Top of MCP

5d ago

See more recommendations