← All Posts

11 Tips For AI Coding With Ralph Wiggum

Matt Pocock

On this page > 2. Start With HITL, Then Go AFK

If you're using AI coding CLIs like Claude Code, Copilot CLI, OpenCode, and Codex, this article is for you.

Most developers use these tools interactively. You give it a task, watch it work, and intervene when it goes off-track. This is "human-in-the-loop" (HITL) coding.

But there's a new approach called Ralph Wiggum. Ralph runs your AI coding CLI in a loop, letting it work autonomously on a list of tasks. You define what needs to be done. Ralph figures out how - and keeps going until it's finished. In other words, it's long-running, autonomous, and unsupervised AFK coding.

This is not a quickstart guide. If you want to get up and running fast, read Getting Started With Ralph.

The 11 Tips

#	Tip	Summary
1	Ralph Is A Loop	What Ralph is and why it works

#	Tip	Summary
2	Start With HITL, Then Go AFK	The two modes of running Ralph
3	Define The Scope	How to specify what "done" looks like
4	Track Ralph's Progress	Using progress files between iterations
5	Use Feedback Loops	Types, tests, and linting as guardrails
6	Take Small Steps	Why smaller tasks produce better code
7	Prioritize Risky Tasks	Tackle hard problems first
8	Explicitly Define Software Quality	Don't let Ralph cut corners
9	Use Docker Sandboxes	Isolate AFK Ralph for safety
10	Pay To Play	Cost considerations and tradeoffs
11	Make It Your Own	Alternative loop types and customization

1. Ralph Is A Loop

AI coding has evolved through distinct phases in the last year or so. Let's briefly define them:

Vibe coding is where you let the AI write code without really checking it. You "vibe" with the AI, accepting its suggestions without scrutiny. It's fast, but the code quality suffers.

Planning is where you ask the AI to plan before it codes. In Claude Code, you can enter plan mode to have the AI explore your codebase and create a plan before writing code. This improves quality, but you're still limited to

what fits in one context window.

Multi-phase plans break large features into phases, each handled in a separate context window. You write a different prompt for each phase: "Implement the database schema," then "Add the API endpoints," then "Build the UI." This scales better, but requires constant human involvement to write each prompt.

Ralph simplifies everything. Instead of writing a new prompt for each phase, you run the same prompt in a loop:

```
# ralph.sh
# Usage: ./ralph.sh <iterations>

set -e

if [ -z "$1" ]; then
  echo "Usage: $0 <iterations>"
  exit 1
fi

# For each iteration, run Claude Code with the following prompt.
# This prompt is basic, we'll expand it later.
for ((i=1; i<=$1; i++)); do
  result=$(docker sandbox run claude -p \
"@some-plan-file.md @progress.txt \
1. Decide which task to work on next. \
This should be the one YOU decide has the highest priority, \
- not necessarily the first in the list. \
2. Check any feedback loops, such as types and tests. \
3. Append your progress to the progress.txt file. \
4. Make a git commit of that feature. \
ONLY WORK ON A SINGLE FEATURE. \
If, while implementing the feature, you notice that all work \
is complete, output <promise>COMPLETE</promise>. \
")

  echo "$result"
```

```
  if [[ "$result" == *"<promise>COMPLETE</promise>"* ]]; then
    echo "PRD complete, exiting."
    exit 0
  fi
done
```

Each iteration:

- Looks at a plan file to see what needs to be done
- Looks at a progress file to see what has already been done
- Decides what to do next
- Explores the codebase
- Implements the feature
- Runs feedback loops (types, linting, tests)
- Commits the code

The key improvement here is that **the agent chooses the task, not you**.

With multi-phase plans, a human writes a new prompt at the start of each phase. With Ralph, the agent picks what to work on next from your PRD. You define the end state. Ralph gets there.

I've used Ralph to add tests to my AI Hero CLI and build features for my course video manager. Others have built entire programming languages with it.

2. Start With HITL, Then Go AFK

There are two ways to run Ralph:

Mode	How It Works	Best For	Script
HITL (human-in-the-loop)	Run once, watch, intervene	Learning, prompt refinement	`ralph-once.sh`
AFK (away from keyboard)	Run in a loop with max iterations	Bulk work, low-risk tasks	`afk-ralph.sh`

For HITL Ralph, keep a `ralph-once.sh` that runs a single iteration. You watch everything it does and step in when needed.

For AFK Ralph, always cap your iterations. Infinite loops are dangerous with stochastic systems. I typically use 5-10 iterations for small tasks, or 30-50 for larger ones.

HITL Ralph resembles pair programming. You and the AI work together, reviewing code as it's created. You can steer, contribute, and share project understanding in real-time.

It's also the best way to learn Ralph. You'll understand what it does, refine your prompt, and build confidence before going hands-off.

Once your prompt is solid, AFK Ralph unlocks real leverage. Set it running, do something else, come back when it's done.

I built a small CLI to ping me on WhatsApp when Ralph finishes. This means much less context switching. I can fully engage with another task. My loops usually take 30-45 minutes, though they can run for hours.

The progression is simple:

- Start with HITL to learn and refine

- Go AFK once you trust your prompt
- Review the commits when you return

3. Define The Scope

Before you let Ralph run, you need to define what "done" looks like. This is a shift from planning to requirements gathering. Instead of specifying each step, you describe the desired end state and let the agent figure out how to get there.

Formats For Defining Scope

There are many ways to define scope for Ralph:

- A markdown list of user stories
- GitHub issues or Linear tasks (more on this later)
- Using beads

One approach I like comes from Anthropic's research on long-running agents. They structure PRD items as JSON with a `passes` field:

```
{
  "category": "functional",
  "description": "New chat button creates a fresh conversation",
  "steps": [
    "Click the 'New Chat' button",
    "Verify a new conversation is created",
    "Check that chat area shows welcome state"
  ],
  "passes": false
}
```

Ralph marks `passes` to `true` when complete. The PRD becomes both scope definition and progress tracker - a living TODO list rather than a waterfall document.

Why Scope Matters

You don't *need* a structured TODO list. You can give Ralph a vague task - "improve this codebase" - and let it track its own progress.

But the vaguer the task, the greater the risk. Ralph might loop forever, finding endless improvements. Or it might take shortcuts, declaring victory before you'd consider the job done:

I ran Ralph to increase test coverage on my AI Hero CLI. The repo had internal commands - marked as internal but still user-facing (I use them). I wanted tests for everything.

After three iterations, Ralph reported: "Done with all user-facing commands." But it had skipped the internal ones entirely. It decided they weren't user-facing and marked them to be ignored by coverage.

The fix? Define exactly what you want covered:

What to Specify	Why It Prevents Shortcuts
Files to include	Ralph won't ignore "edge case" files
Stop condition	Ralph knows when "complete" actually means complete
Edge cases	Ralph won't decide certain things don't count

Adjusting PRDs Mid-Flight

One benefit of this approach: you can adjust while Ralph is running.

- Already implemented but wrong? Set `passes` back to `false`, add notes, rerun.
- Missing a feature? Add a new PRD item even mid-loop.

You're not editing a linear multi-phase plan. You're describing a different end state. Ralph will get there.

As long as scope and stop condition are explicit, Ralph will know when to emit `<promise>COMPLETE</promise>`.

Try It Out

Use plan mode, and create a `prd.json` file for your next feature. Use this prompt to generate structured PRD items:

```
Convert my feature requirements into structured PRD items.
Each item should have: category, description, steps to verify, and passes: false.
Format as JSON. Be specific about acceptance criteria.
```

4. Track Ralph's Progress

Every Ralph loop I run emits a `progress.txt` file, committed directly to the repo. I took this inspiration from Anthropic's article on long-running agent harnesses.

This addresses a core challenge: AI agents are like super-smart experts who forget everything between tasks. Each new context window starts

fresh. Without a progress file, Ralph must explore the entire repo to understand the current state.

A progress file short-circuits that exploration. Ralph reads it, sees what's done, and jumps straight into the next task.

What Goes In The Progress File

Keep it simple and concise:

- Tasks completed in this session
- Decisions made and why
- Blockers encountered
- Files changed

You can also include the PRD item that was just completed, any architectural decisions, and notes for the next iteration.

Why Commits Matter

Ralph should commit after each feature. This gives future iterations:

- A clean `git log` showing what changed
- The ability to `git diff` against previous work
- A rollback point if something breaks

The combination of progress file plus git history gives Ralph full context without burning tokens on exploration.

Cleanup

Don't keep `progress.txt` forever. Once your sprint is done, delete it. It's session-specific, not permanent documentation.

Try It Out

Add progress tracking to your Ralph prompt:

```
After completing each task, append to progress.txt:
- Task completed and PRD item reference
- Key decisions made and reasoning
- Files changed
- Any blockers or notes for next iteration
Keep entries concise. Sacrifice grammar for the sake of concision. This file help
```

5. Use Feedback Loops

Ralph's success depends on feedback loops. The more loops you give it, the higher quality code it produces.

Types of Feedback Loops

Feedback Loop	What It Catches
TypeScript types	Type mismatches, missing props
Unit tests	Broken logic, regressions
Playwright MCP server	UI bugs, broken interactions
ESLint / linting	Code style, potential bugs

Feedback Loop	What It Catches
Pre-commit hooks	Blocks bad commits entirely

The best setup blocks commits unless everything passes. Ralph can't declare victory if the tests are red.

Why Feedback Loops Matter

Great programmers don't trust their own code. They don't trust external libraries. They especially don't trust their colleagues. Instead, they build automations and checks to verify what they ship.

This humility produces better software. The same applies to AI agents.

Every tip in this article works for human developers too. Feedback loops, small steps, explicit scope - these aren't AI-specific techniques. They're just good engineering. Ralph makes them non-negotiable.

Try It Out

Add explicit feedback loop requirements to your Ralph prompt:

```
Before committing, run ALL feedback loops:
1. TypeScript: npm run typecheck (must pass with no errors)
2. Tests: npm run test (must pass)
3. Lint: npm run lint (must pass)
Do NOT commit if any feedback loop fails. Fix issues first.
```

6. Take Small Steps

The rate at which you can get feedback is your speed limit. Never outrun your headlights.

Humans doing a big refactor might bite off a huge chunk and roll through it. Tests, types, and linting stay red for hours. Breaking work into smaller chunks means tighter feedback loops - less work before you receive feedback.

The same applies to Ralph, but with an additional constraint: context windows are limited, and LLMs get worse as they fill up. This is called context rot - the longer you go, the stupider the output.

The Tradeoff

Each Ralph iteration has startup costs. Ralph must pick a task, explore the repo, and gather context. These tokens are spent per-loop.

If you're doing a large refactor, you don't want Ralph renaming one variable per iteration. But:

- Larger tasks mean less frequent feedback
- More context means lower quality code
- Smaller tasks mean higher quality, but slower progress

Sizing Your PRD Items

For AFK Ralph, keep PRD items small. You want the agent on top form when you're not watching.

For HITL Ralph, you can make items slightly larger to see progress faster.

But even then, bias small.

A refactor item might be as simple as: "Change one function's parameters. Verify tests and types pass."

In your prompt, guide Ralph on step size. My tendency: small steps. Code quality over speed - especially when AFK, where speed matters less anyway.

Try It Out

Add step-size guidance to your Ralph prompt:

```
Keep changes small and focused:
- One logical change per commit
- If a task feels too large, break it into subtasks
- Prefer multiple small commits over one large commit
- Run feedback loops after each change, not at the end
Quality over speed. Small steps compound into big progress.
```

7. Prioritize Risky Tasks

Ralph chooses its own tasks. Without explicit guidance, it will often pick the first item in the list or whatever seems easiest to implement.

This mirrors human behavior. Developers love quick wins. But seasoned engineers know you should nail down the hard stuff first, before the easy work buries you in technical debt.

Spikes And Integration

Focus on spikes - things you don't know how they'll turn out. Build features end-to-end rather than layer by layer. Integrate early.

If you have modules that need to work together, tell Ralph to integrate them first. Don't wait until the end of your sprint to discover they don't fit.

Task Type	Priority	Why
Architectural work	High	Decisions cascade through entire codebase
Integration points	High	Reveals incompatibilities early
Unknown unknowns	High	Better to fail fast than fail late
UI polish	Low	Can be parallelized later
Quick wins	Low	Easy to slot in anytime

HITL For Risky Tasks

Risky tasks need more human involvement. Use HITL Ralph for early architectural decisions - the code from these tasks stays forever, and any shortcuts here will cascade through the entire project.

Save AFK Ralph for when the foundation is solid. Once the architecture is proven and the risky integrations work, you can let Ralph run unsupervised on the lower-risk tasks.

Try It Out

Add prioritization guidance to your Ralph prompt:

```
When choosing the next task, prioritize in this order:
```

```
1. Architectural decisions and core abstractions
2. Integration points between modules
3. Unknown unknowns and spike work
4. Standard features and implementation
5. Polish, cleanup, and quick wins
Fail fast on risky work. Save easy wins for later.
```

8. Explicitly Define Software Quality

Not all repos are made alike. A lot of code out there is prototype code - demos, short-lived experiments, client pitches. Different repos have different bars for quality.

The agent doesn't know what kind of repo it's in. It doesn't know if this is a throwaway prototype or production code that will be maintained for years. You need to tell it explicitly.

What To Communicate

Repo Type	What To Say	Expected Behavior
Prototype	"This is a prototype. Speed over perfection."	Takes shortcuts, skips edge cases
Production	"Production code. Must be maintainable."	Follows best practices, adds tests
Library	"Public API. Backward compatibility matters."	Careful about breaking changes

Put this in your AGENTS.md file, your skills, or directly in your prompt.

The Repo Wins

Your instructions compete with your codebase. When Ralph explores your repo, it sees two sources of truth: what you told it to do and what you actually did. One is a few lines of instruction. The other is thousands of lines of evidence.

You can write "never use `any` types" in your prompt. But if Ralph sees `any` throughout your existing code, it will follow the codebase, not your instructions.

Agents amplify what they see. Poor code leads to poorer code. Low-quality tests produce unreliable feedback loops.

This is software entropy - the tendency of codebases to deteriorate over time. Ralph accelerates this. A human might commit once or twice a day. Ralph can pile dozens of commits into a repo in hours. If those commits are low quality, entropy compounds fast.

This means you need to:

- Keep your codebase clean before letting Ralph loose
- Use feedback loops (linting, types, tests) to enforce standards
- Make quality expectations explicit and visible

Try It Out

Add quality expectations to your `AGENTS.md` or Ralph prompt:

```
This codebase will outlive you. Every shortcut you take becomes
someone else's burden. Every hack compounds into technical debt
```

```
that slows the whole team down.

You are not just writing code. You are shaping the future of this
project. The patterns you establish will be copied. The corners
you cut will be cut again.

Fight entropy. Leave the codebase better than you found it.
```

9. Use Docker Sandboxes

AFK Ralph needs permissions to edit files, run commands, and commit code. What stops it from running `rm -rf ~`? You're away from the keyboard, so you're not going to be able to intervene.

Docker sandboxes are the simplest solution:

```
docker sandbox run claude
```

This runs Claude Code inside a container. Your current directory is mounted, but nothing else. Ralph can edit project files and commit - but can't touch your home directory, SSH keys, or system files.

The tradeoff: your global AGENTS.md and user skills won't be loaded. For most Ralph loops, this is fine.

For HITL Ralph, sandboxes are optional - you're watching. For AFK Ralph, especially overnight loops, they're essential insurance against runaway agents.

10. Pay To Play

One question I get a lot is, "How much will this cost?" Surely running AFK Ralph overnight is a way to rack up enormous bills?

I never feel comfortable giving financial advice, especially to folks in low-income countries. But Ralph is completely configurable to how much you want to spend.

HITL Is Still Worth It

If you never run AFK Ralph, HITL Ralph still has big benefits over multi-phase planning. Running the same prompt over and over feels nicer than specifying a different prompt for each phase.

Approach	Effort Per Phase	Best For
Multi-phase plans	Write new prompt	One-off large tasks
HITL Ralph	Rerun same prompt	Learning, refinement
AFK Ralph	Set and forget	Bulk work, automation

I'm on the Anthropic 5x Max plan at around £90/month. I've run AFK Ralph a few times, but most of my usage is HITL.

Why Not Local Models?

I don't think open source models you can run on your laptop are good enough for Ralph yet. They require powerful GPUs, and the output quality isn't there. In AI coding, you have to pay to play.

The Golden Age

But we need to contextualize this. For the next couple of years, we're in a golden age where you can do magical things with AI faster than humans - but the market still pays human wages. The market hasn't adjusted to the fact that we all have access to extremely powerful AI coding tools.

Yes, you have to pay. But the rewards are there if you're willing to claim them.

11. Make It Your Own

Ralph is just a loop. That simplicity makes it infinitely configurable. Here are some ways to make it your own:

Swap The Task Source

The examples in this article use a local `prd.json`. But Ralph can pull tasks from anywhere:

Task Source	How It Works
GitHub Issues	Ralph picks an issue, implements it
Linear	Ralph pulls from your sprint
Beads	Ralph works through a beadfile

The key insight stays the same: the agent chooses the task, not you. You're just changing where that list lives.

Change The Output

Instead of committing directly to main, each Ralph iteration could:

- Create a branch and open a PR
- Add comments to existing issues
- Update a changelog or release notes

This is useful when you have a backlog of issues that need to become PRs. Ralph triages, implements, and opens the PR. You review when you're ready.

Alternative Loop Types

Ralph doesn't need to work through a feature backlog. Some loops I've been experimenting with:

Test Coverage Loop: Point Ralph at your coverage metrics. It finds uncovered lines, writes tests, and iterates until coverage hits your target. I used this to take AI Hero CLI from 16% to 100% coverage.

Duplication Loop: Hook Ralph up to `jscpd` to find duplicate code. Ralph identifies clones, refactors into shared utilities, and reports what changed.

Linting Loop: Feed Ralph your linting errors. It fixes them one by one, running the linter between iterations to verify each fix.

Entropy Loop: Ralph scans for code smells - unused exports, dead code, inconsistent patterns - and cleans them up. Software entropy in reverse.

Any task that can be described as "look at repo, improve something, report

findings" fits the Ralph pattern. The loop is the same. Only the prompt changes.

Try It Out

Try one of these alternative loop prompts:

```
# Test Coverage Loop
@coverage-report.txt
Find uncovered lines in the coverage report.
Write tests for the most critical uncovered code paths.
Run coverage again and update coverage-report.txt.
Target: 80% coverage minimum.
```

```
# Linting Loop
Run: npm run lint
Fix ONE linting error at a time.
Run lint again to verify the fix.
Repeat until no errors remain.
```

```
# Entropy Loop
Scan for code smells: unused exports, dead code, inconsistent patterns.
Fix ONE issue per iteration.
Document what you changed in progress.txt.
```

I look forward to seeing your own Ralph Wiggums - fingers up noses, flying through windows, eating paste, and shipping code.

Want to learn more about Ralph? I'll be publishing a lot more about autonomous AI coding on my newsletter. Sign up to get notified when new

articles drop.

Join over 50,000 Developers Becoming AI Heroes

Subscribe to be the first to learn about AI Hero releases, updates, and special discounts for AI Heroes.

First Name

Name

Email*

you@example.com

Subscribe

I respect your privacy. Unsubscribe at any time.

Share Copy URL

Up Next

Your App Is Only As Good As Its Evals →

Log in to save progress