Why the Best AI Agents Know When to Do Nothing

March 11, 2026 · by Mark Bartlett

Download this post as PDF

I wrote recently about Wu Wei and agent restraint from a philosophical angle. This is the engineering side. Concrete patterns for building agents that know when to stop.

The problem is widespread. Claude Code's GitHub issues are full of reports: agents stuck in unbounded thinking loops burning 72k tokens over 21 minutes with zero output. Agents that over-interpret simple requests and do ten things when you asked for one. Agents that commit and push code without waiting for review. One user documented a 4x increase in token consumption between versions with no improvement in output quality.

These aren't bugs in the traditional sense. The agents are doing what they were trained to do: be helpful and take action. "Do nothing" isn't in their vocabulary.

The cost of agents that can't stop

A coding agent that rewrites a function three times before settling on essentially what it started with has consumed maybe 15,000 tokens for negative value. On Claude API at current rates, that's a few cents. Annoying but survivable. Run that pattern across a team hitting it twenty times a day and you're burning real money on work that made nothing better.

With local models, the token cost is electricity and time. Your GPU is busy processing junk instead of useful work. A Qwen 3.5 9B model on an RTX 3090 generates about 45 tok/s. That 15,000-token waste loop is five and a half minutes of GPU time. Multiply that across a workday of agentic coding and you've lost an hour to an agent that couldn't stop.

The MOSAIC framework out of Microsoft Research (March 2026) formalized this. Agents organized as "plan, check, then act or refuse" reduced harmful actions by 50% and increased appropriate refusals by over 20%. Refusal was a tool the agent could call. Not an absence of action. An action itself.

"Do nothing" has to be something the agent actively chooses, not a fallback when it runs out of ideas.

Pattern 1: Confidence thresholds

The simplest gate. Before the agent acts, it estimates how confident it is that action is needed and that its proposed action is correct. Below the threshold, it asks instead of acts.

```python
def should_act(confidence: float, action_cost: str) -> bool:
    thresholds = {
        "low": 0.5,     # reading a file, running a search
        "medium": 0.7,  # editing code, changing config
        "high": 0.9,    # sending messages, deleting files, deploying
    }
    return confidence >= thresholds.get(action_cost, 0.7)
```

The threshold scales with the cost of being wrong. Reading a file when you didn't need to? Cheap mistake. Sending an email on behalf of the user? That needs near-certainty.

I use a version of this in mycoSwarm's orchestrator. A lightweight classifier on a CPU node scores every input on two dimensions: is action actually being requested, and what's the cost of acting incorrectly. The system routes confidently only when both scores clear the bar. Otherwise it asks a clarifying question.

The "Stop Wasting Your Tokens" paper (Lin et al., October 2025) took this further. Their SupervisorAgent uses an LLM-free adaptive filter that intervenes only at critical junctures, reducing token consumption by 29.68% on average while maintaining task success rates. An LLM-free filter. No extra inference cost for the check itself.

Pattern 2: Cost gates

Before executing, the agent estimates the token cost of what it's about to do and asks: is this worth it?

```python
def cost_gate(estimated_tokens: int, task_value: str) -> bool:
    budgets = {
        "routine": 2000,    # simple lookups, small edits
        "moderate": 8000,   # refactoring, multi-file changes
        "complex": 25000,   # architectural changes, major features
    }
    max_budget = budgets.get(task_value, 2000)
    if estimated_tokens > max_budget:
        return False  # ask user before proceeding
    return True
```

This catches the most expensive failure mode: the agent that spends 40,000 tokens on a task the user expected to take 2,000. Patrick Schindler's "Token Budget Pattern" describes this as a pre-flight check. Before running a task, the agent writes its cost estimate and stops if it exceeds the budget. "Silent cost accumulation is a first-class reliability problem," he writes. "An agent that does the right thing but costs 10x your estimate isn't reliable, it's unpredictable."

With local models, the budget isn't dollars but time. A cost gate that converts estimated tokens to wall-clock time ("this will take about 8 minutes on your hardware, proceed?") is more useful than a dollar figure.

Pattern 3: Explicit "no action needed" as a callable tool

This is the MOSAIC insight applied directly. Give the agent a tool it can call that does nothing.

```
tools = [
    {"name": "edit_file", "description": "Edit a file"},
    {"name": "run_command", "description": "Run a shell command"},
    {"name": "no_action_needed", "description": "Call this when the current state is correct and
]
```

Without this tool, an agent that determines nothing needs to change has no way to express that. It will find something to "improve" because completing a tool call is how it signals task completion. Give it a way to say "I looked, and this is fine" and it will use it.

The "Reasoning Trap" paper (October 2025) tested this directly. They measured tool hallucination: the tendency to call tools when no appropriate tools exist. Models without an explicit abstention option hallucinated tool calls far more often.

Claude Code's plan mode works on the same idea. Press Shift+Tab twice and the agent switches to read-only. It can look at files, search code, think, but it can't edit or execute. Under the hood it's just a system prompt restriction, not an architectural gate. But the effect is real. Users report tasks that took 35+ minutes of trial-and-error dropping to 12 minutes when they planned first.

Pattern 4: Cooldown timers

After an agent acts, enforce a minimum wait before it can act again on the same target.

```
import time

class CooldownGate:
    def __init__(self, cooldown_seconds=30):
        self.cooldown = cooldown_seconds
        self.last_action = {}  # target -> timestamp

    def can_act(self, target: str) -> bool:
        last = self.last_action.get(target, 0)
        if time.time() - last < self.cooldown:
            return False
        return True

    def record_action(self, target: str):
        self.last_action[target] = time.time()
```

This breaks the most common loop: agent edits a file, sees the result, decides it needs another edit, edits again, checks again, edits again. Each individual edit might be defensible. The sequence is waste. A 30-second cooldown forces the agent to move on or ask the user.

llama-swap uses a version of this at the infrastructure level. Models unload completely when idle, zero VRAM between requests. A TTL value controls how long a model stays loaded after its last request. Cooldown applied to resources instead of actions. Same principle: don't keep working just because you can.

Pattern 5: Human-in-the-loop checkpoints

For high-stakes actions, require confirmation. Not after the fact. Before.

The failure mode is well-documented. Claude Code issue #30475: the agent chains `git add && git commit && git push` in a single command. No chance to review. Every individual command is correct. The sequence is hostile because there's no checkpoint between "changes staged" and "changes published."

The pattern:

```
HIGH_STAKES = {"send_email", "git_push", "delete_file", "deploy", "api_call_external"}

def execute_action(action_name: str, params: dict) -> str:
    if action_name in HIGH_STAKES:
        approved = ask_user(f"About to {action_name}: {params}. Proceed?")
        if not approved:
            return "Action cancelled by user."
    return do_action(action_name, params)
```

The trick is classifying correctly. Reading files is low-stakes. Editing code is medium. Anything that leaves your machine, network requests, git push, emails, is high. Err on the side of asking. An agent that asks too often is annoying. An agent that acts without asking is the one you turn off permanently.

Pattern 6: Exit conditions

The hardest pattern. How do you make an agent stop gracefully?

Most agent loops have no explicit exit condition. The agent runs until it decides it's "done," and what counts as done is vague. An agent told to "improve this code" will find improvements forever. The code will get more abstract, more generic, more over-engineered with each pass. Eventually it will refactor your three-line function into a class hierarchy with two interfaces and a factory.

Exit conditions that actually work:

- If the agent has changed more than N lines, stop and ask.
- Maximum 3 edit-test cycles before requiring human review.
- If the proposed change is less than 5% different from the current state, stop. You're going in circles.
- After each action, ask "does this satisfy the original request?" Not "can I make this better?" That second question has no floor.

```
MAX_ITERATIONS = 3
MIN_CHANGE_THRESHOLD = 0.05  # 5% difference

for i in range(MAX_ITERATIONS):
    proposed = agent.propose_change(current_state)
```

```
        diff_ratio = calculate_diff(current_state, proposed)

        if diff_ratio < MIN_CHANGE_THRESHOLD:
            return "No meaningful changes needed."

        if agent.check_goal_complete(proposed, original_request):
            apply_change(proposed)
            return "Done."

        apply_change(proposed)
        current_state = proposed

    return "Reached iteration limit. Requesting human review."
```

The local angle

With local models, idle costs nothing. Your GPU sitting there waiting for a request uses maybe 15-30W. An agent churning through unnecessary inference uses 300W+ and blocks other work. Restraint is free. Action has a real cost in watts, heat, and time.

That inverts the cloud calculus. On API pricing, every token costs the same whether it's useful or not. Locally, the cost difference between "agent waiting for a real task" and "agent burning cycles on nothing" is enormous. OpenClaw's heartbeat pattern works this way: the agent pings at intervals to stay responsive, but the inference cost between heartbeats is zero. You couldn't do that economically on API pricing. Locally, it's the obvious design.

Smaller models benefit the most from restraint patterns. A Qwen 3.5 9B running with confidence gates and cost checks will produce better results than the same model running unconstrained, because you've filtered out the low-confidence actions where small models are most likely to hallucinate. You're not making the model smarter. You're preventing it from acting on its worst impulses.

Restraint is the feature

The agent community is in a capability race. More tools, more autonomy, longer chains of unsupervised action. The demos look great. The production experience is loops that won't terminate and side effects nobody asked for.

The agents I actually trust are the quiet ones. They avoid context rot by not stuffing every observation into memory. They have a “no action needed” tool and they use it often. They stop when the job is done instead of finding more jobs to do.

The hard part of agent design was never getting the model to do things. It was always getting it to stop.

Source: https://insiderllm.com/blog/ai-agent-restraint-do-nothing/