Effective harnesses for long-running agents

Agents still face challenges working across many context windows. We looked to human engineers for inspiration in creating a more effective harness for long-running agents.

Published Nov 26, 2025

As AI agents become more capable, developers are increasingly asking them to take on complex tasks requiring work that spans hours, or even days. However, getting agents to make consistent progress across multiple context windows remains an open problem.

The core challenge of long-running agents is that they must work in discrete sessions, and each new session begins with no memory of what came before. Imagine a software project staffed by engineers working in shifts, where each new engineer arrives with no memory of what happened on the previous shift. Because context windows are limited, and because most complex projects cannot be completed within a single window, agents need a way to bridge the gap between coding sessions.

We developed a two-fold solution to enable the Claude Agent SDK to work effectively across many context windows: an **initializer agent** that sets up the environment on the

first run, and a **coding agent** that is tasked with making incremental progress in every session, while leaving clear artifacts for the next session. You can find code examples in the accompanying quickstart.

The long-running agent problem

The Claude Agent SDK is a powerful, general-purpose agent harness adept at coding, as well as other tasks that require the model to use tools to gather context, plan, and execute. It has context management capabilities such as compaction, which enables an agent to work on a task without exhausting the context window. Theoretically, given this setup, it should be possible for an agent to continue to do useful work for an arbitrarily long time.

However, compaction isn't sufficient. Out of the box, even a frontier coding model like Opus 4.5 running on the Claude Agent SDK in a loop across multiple context windows will fall short of building a production-quality web app if it's only given a high-level prompt, such as "build a clone of claude.ai."

Claude's failures manifested in two patterns. First, the agent tended to try to do too much at once—essentially to attempt to one-shot the app. Often, this led to the model running out of context in the middle of its implementation, leaving the next session to start with a feature half-implemented and undocumented. The agent would then have to guess at what had happened, and spend substantial time trying to get the basic app working again. This happens even with compaction, which doesn't always pass perfectly clear instructions to the next agent.

A second failure mode would often occur later in a project. After some features had already been built, a later agent instance would look around, see that progress had been made, and declare the job done.

This decomposes the problem into two parts. First, we need to set up an initial environment that lays the foundation for *all* the features that a given prompt requires, which sets up the agent to work step-by-step and feature-by-feature. Second, we should prompt each agent to make incremental progress towards its goal while also leaving the environment in a clean state at the end of a session. By "clean state" we mean the kind of code that would be appropriate for merging to a main branch: there are no major bugs, the code is orderly and well-documented, and in general, a developer could easily begin work on a new feature without first having to clean up an unrelated mess.

When experimenting internally, we addressed these problems using a two-part solution:

1. Initializer agent: The very first agent session uses a specialized prompt that asks the model to set up the initial environment: an `init.sh` script, a claude-progress.txt file that keeps a log of what agents have done, and an initial git commit that shows what files were added.
2. Coding agent: Every subsequent session asks the model to make incremental progress, then leave structured updates.[1]

The key insight here was finding a way for agents to quickly understand the state of work when starting with a fresh context window, which is accomplished with the claude-progress.txt file alongside the git history. Inspiration for these practices came from knowing what effective software engineers do every day.

Environment management

In the updated Claude 4 prompting guide, we shared some best practices for multi-context window workflows, including a harness structure that uses "a different prompt for the very first context window." This "different prompt" requests that the initializer agent set up the environment with all the necessary context that future coding agents will need to work effectively. Here, we provide a deeper dive on some of the key components of such an environment.

Feature list

To address the problem of the agent one-shotting an app or prematurely considering the project complete, we prompted the initializer agent to write a comprehensive file of feature requirements expanding on the user's initial prompt. In the claude.ai clone example, this meant over 200 features, such as "a user can open a new chat, type in a query, press enter, and see an AI response." These features were all initially marked as "failing" so that later coding agents would have a clear outline of what full functionality looked like.

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

Copy

We prompt coding agents to edit this file only by changing the status of a passes field, and we use strongly-worded instructions like "It is unacceptable to remove or edit tests because this could lead to missing or buggy functionality." After some experimentation, we landed on using JSON for this, as the model is less likely to inappropriately change or overwrite JSON files compared to Markdown files.

Incremental progress

Given this initial environment scaffolding, the next iteration of the coding agent was then asked to work on only one feature at a time. This incremental approach turned out to be critical to addressing the agent's tendency to do too much at once.

Once working incrementally, it's still essential that the model leaves the environment in a clean state after making a code change. In our experiments, we found that the best way to elicit this behavior was to ask the model to commit its progress to git with descriptive commit messages and to write summaries of its progress in a progress file. This allowed the model to use git to revert bad code changes and recover working states of the code base.

These approaches also increased efficiency, as they eliminated the need for an agent to have to guess at what had happened and spend its time trying to get the basic app working again.

Testing

One final major failure mode that we observed was Claude's tendency to mark a feature as complete without proper testing. Absent explicit prompting, Claude tended to make code changes, and even do testing with unit tests or `curl` commands against a development server, but would fail recognize that the feature didn't work end-to-end.

In the case of building a web app, Claude mostly did well at verifying features end-to-end once explicitly prompted to use browser automation tools and do all testing as a human user would.

Screenshots taken by Claude through the Puppeteer MCP server as it tested the claude.ai clone.

Providing Claude with these kinds of testing tools dramatically improved performance, as the agent was able to identify and fix bugs that weren't obvious from the code alone.

Some issues remain, like limitations to Claude's vision and to browser automation tools making it difficult to identify every kind of bug. For example, Claude can't see browser-native alert modals through the Puppeteer MCP, and features relying on these modals tended to be buggier as a result.

Getting up to speed

With all of the above in place, every coding agent is prompted to run through a series of steps to get its bearings, some quite basic but still helpful:

1. *Run* `pwd` *to see the directory you're working in. You'll only be able to edit files in this directory.*
2. *Read the git logs and progress files to get up to speed on what was recently worked on.*
3. *Read the features list file and choose the highest-priority feature that's not yet done to work on.*

This approach saves Claude some tokens in every session since it doesn't have to figure out how to test the code. It also helps to ask the initializer agent to write an init.sh script that can run the development server, and then run through a basic end-to-end test before implementing a new feature.

In the case of the claude.ai clone, this meant that the agent always started the local development server and used the Puppeteer MCP to start a new chat, send a message, and receive a response. This ensured that Claude could quickly identify if the app had been left in a broken state, and immediately fix any existing bugs. If the agent had instead started implementing a new feature, it would likely make the problem worse.

Given all this, a typical session starts off with the following assistant messages:

```
[Assistant] I'll start by getting my bearings and understanding the
current state of the project.
[Tool Use] <bash - pwd>
[Tool Use] <read - claude-progress.txt>
[Tool Use] <read - feature_list.json>
[Assistant] Let me check the git log to see recent work.
[Tool Use] <bash - git log --oneline -20>
[Assistant] Now let me check if there's an init.sh script to
restart the servers.
<Starts the development server>
[Assistant] Excellent! Now let me navigate to the application and
verify that some fundamental features are still working.
<Tests basic functionality>
[Assistant] Based on my verification testing, I can see that the
fundamental functionality is working well. The core chat features,
theme switching, conversation loading, and error handling are all
functioning correctly. Now let me review the tests.json file more
comprehensively to understand what needs to be implemented next.
<Starts work on a new feature>
```

Copy

Agent failure modes and solutions

Problem	Initializer Agent Behavior	Coding Agent Behavior
Claude declares victory on the entire project too early.	Set up a feature list file: based on the input spec, set up a structured JSON file with a list of end-to-end feature descriptions.	Read the feature list file at the beginning of a session. Choose a single feature to start working on.

Claude leaves the environment in a state with bugs or undocumented progress.	An initial git repo and progress notes file is written.
Start the session by reading the progress notes file and git commit logs, and run a basic test on the development server to catch any undocumented bugs. End the session by writing a git commit and progress update.	
Claude marks features as done prematurely.	Set up a feature list file.
Self-verify all features. Only mark features as “passing” after careful testing.	
Claude has to spend time figuring out how to run the app.	Write an `init.sh` script that can run the development server.
Start the session by reading `init.sh`.	

Summarizing four common failure modes and solutions in long-running AI agents.

Future work

This research demonstrates one possible set of solutions in a long-running agent harness to enable the model to make incremental progress across many context windows. However, there remain open questions.

Most notably, it’s still unclear whether a single, general-purpose coding agent performs best across contexts, or if better performance can be achieved through a multi-agent architecture. It seems reasonable that specialized agents like a testing agent, a quality assurance agent, or a code cleanup agent, could do an even better job at sub-tasks across the software development lifecycle.

Additionally, this demo is optimized for full-stack web app development. A future direction is to generalize these findings to other fields. It’s likely that some or all of these lessons can be applied to the types of long-running agentic tasks required in, for example, scientific research or financial modeling.

Acknowledgements

Written by Justin Young. Special thanks to David Hershey, Prithvi Rajasakeran, Jeremy Hadfield, Naia Bouscal, Michael Tingley, Jesse Mu, Jake Eaton, Marius Buleandara, Maggie Vo, Pedram Navid, Nadine Yasser, and Alex Notov for their contributions.

This work reflects the collective efforts of several teams across Anthropic who made it possible for Claude to safely do long-horizon autonomous software engineering, especially the code RL & Claude Code teams. Interested candidates who would like to contribute are welcome to apply at anthropic.com/careers.

Footnotes

1. We refer to these as separate agents in this context only because they have different initial user prompts. The system prompt, set of tools, and overall agent harness was otherwise identical.

Opus

Sonnet

Haiku

Solutions

AI agents

Code modernization

Coding

Customer support

Education

Financial services

Government

Healthcare

Life sciences

Nonprofits

Claude Developer Platform

Overview

Developer docs

Pricing

Regional Compliance

Amazon Bedrock

Google Cloud’s Vertex AI

Console login

Learn

Blog

Claude partner network

Connectors

Courses

Customer stories

Engineering at Anthropic

Events

Powered by Claude

Service partners

Startups program

Tutorials

Use cases

Company

Anthropic

Careers

Economic Futures

Research

News

Claude's Constitution

Responsible Scaling Policy

Security and compliance

Transparency

Help and security

Availability

Status

Support center

Terms and policies

Privacy choices

Privacy policy