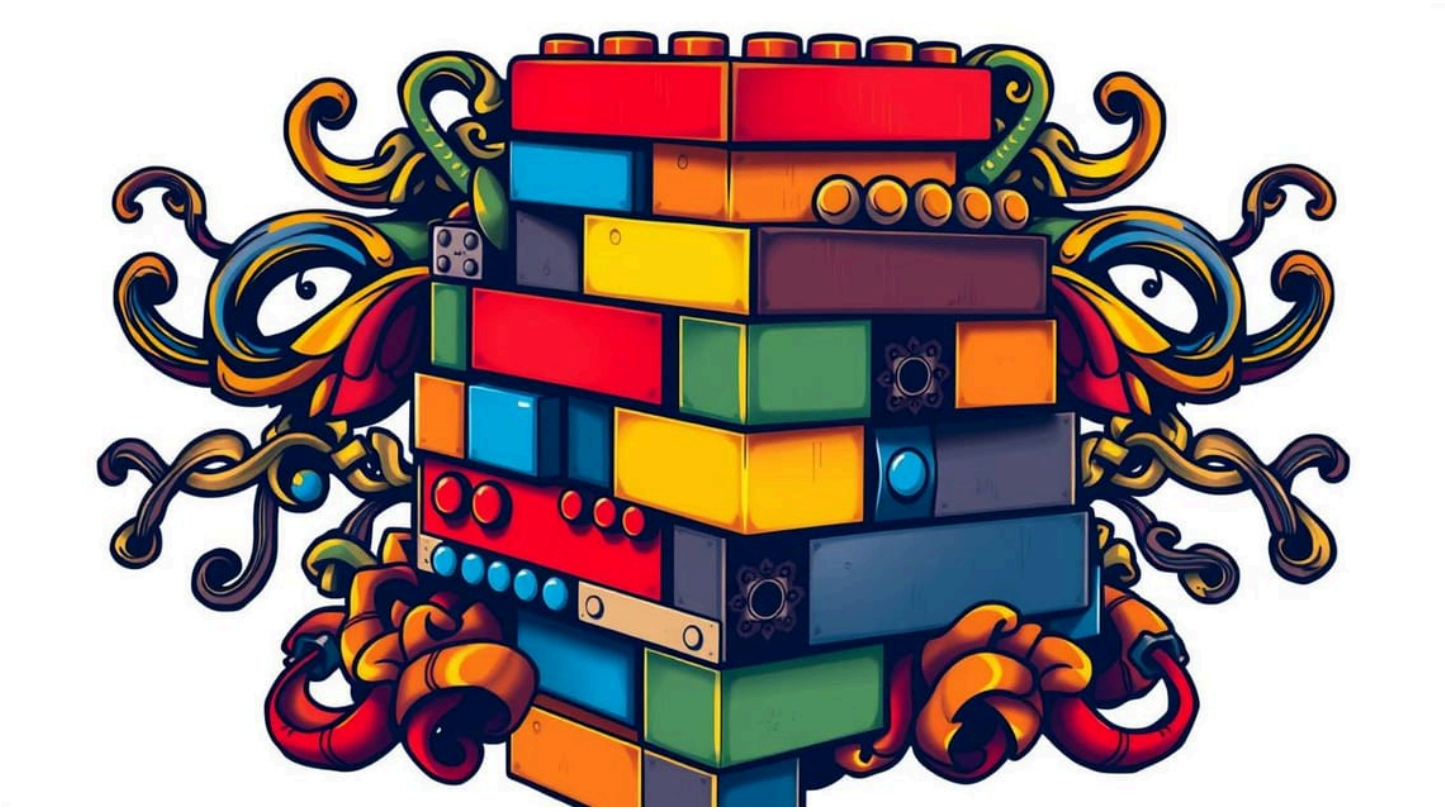
09 FEB 2025 • 10 MIN READ • AI

You are using Cursor AI incorrectly...



I recently shipped a follow-up blog post to this one; this post remains true. You'll need to know this to be able to drive the N-factor of weeks of co-worker output in hours technique as detailed at https://ghuntley.com/specs

I'm hesitant to give this advice away for free, but I'm gonna push past it and share it anyway. You're using the Cursor incorrectly.

Over the last few weeks I've been doing /zooms with software engineers - from entry level, to staff level and all the way up to principal level.

Here's what I've seen:

- Using Cursor as a replacement for Google Search.
- Underspecification of prompts, not knowing how to drive outcomes and using low-level thinking of "implement XYZ, please".
- Treating Cursor as if it is an IDE, instead of it being an autonomous agent.
- Blissful unawareness of the concept that you can program LLM outcomes.
- Unnecessary usage of pleasantries ("please" and "can you") with it as if it were a human. If it fucks up, swear at it - go all caps and call it a clown. It soothes the soul.

Okay, well that last point - it doesn't really change the outcome of Cursor so let's focus on the other points....

Cursor has a pretty powerful feature called Cursor Rules and it's a killer feature that is being slept on/is misunderstood. A quick scour of GitHub for implementations and scouting the community forums reveals that people are using them incorrectly - they all roughly look like this...

```
# WordPress PHP Guzzle Gutenberg .cursorrules prompt file

Author: <redacted>

## What you can build
E-commerce Store Integration Plugin: Create a WordPress plugin that integrate

## Benefits


## Synopsis
WordPress developers can create a plugin that integrates external APIs using

## Overview of .cursorrules prompt
The .cursorrules file provides guidelines for developing a WordPress plugin t
```

Instead of approaching Cursor from the angle of "implement XYZ of code" you should instead be thinking of ***building out a "stdlib" (standard library) of thousands of prompting rules and then composing them together like unix pipes.***

The first rule that every engineering project should have is a function that describes where to store the rules...

```
---
description: Cursor Rules Location
globs: *.mdc
---
# Cursor Rules Location

Rules for placing and organizing Cursor rule files in the repository.

<rule>
name: cursor_rules_location
description: Standards for placing Cursor rule files in the correct directory
filters:
  # Match any .mdc files
  - type: file_extension
    pattern: "\\.mdc$"
  # Match files that look like Cursor rules
  - type: content
    pattern: "(?s)<rule>.*?</rule>"
  # Match file creation events
  - type: event
    pattern: "file_create"

actions:
  - type: reject
    conditions:
      - pattern: "^(?!\\.\\/\\.cursor\\/rules\\/.*\\.mdc$)"
        message: "Cursor rule files (.mdc) must be placed in the .cursor/rule

  - type: suggest
    message: |
      When creating Cursor rules:

      1. Always place rule files in PROJECT_ROOT/.cursor/rules/:
         ```
         .cursor/rules/
         ├── your-rule-name.mdc
         ├── another-rule.mdc
         └── ...
         ```
```

```
      2. Follow the naming convention:
         - Use kebab-case for filenames
         - Always use .mdc extension
         - Make names descriptive of the rule's purpose

      3. Directory structure:
         ```
         PROJECT_ROOT/
         ├── .cursor/
         │   └── rules/
         │   ├── your-rule-name.mdc
         │   └── ...
         └── ...
         ```

      4. Never place rule files:
         - In the project root
         - In subdirectories outside .cursor/rules
         - In any other location

examples:
  - input: |
      # Bad: Rule file in wrong location
      rules/my-rule.mdc
      my-rule.mdc
      .rules/my-rule.mdc

      # Good: Rule file in correct location
      .cursor/rules/my-rule.mdc
    output: "Correctly placed Cursor rule file"

metadata:
  priority: high
  version: 1.0
</rule>
```

Now, you might be wondering why. Ah. That's because people are missing out on the fact ***that you can ask Cursor to write rules***. To build out your "stdlib" you are going to asking Cursor to *write rules and update rules with learnings as if your career depended upon it*.

The foundational LLM models right now are what I'd estimate to be at circa 45% accuracy and require frequent steering. When doing a session in fully automated YOLO mode, my instructions to the composer roughly follow the following steps:

- A lengthy discussion about requirements and listing the requirements out in numbered bullet points so that I can cite the specific requirement when something needs changing or if something goes wrong.
- Asking cursor to write the requirements to a file, that I can reinject back in to the context window if required.
- Attaching the @file and @file-test into the context. Specifically instructing Cursor to "inspect and describe the file"
- Asking the agent to implement the "XYZ" requirement, author tests and documentation.
- Run builds and tests after each change.
- Perform a git commit (via a configured rule) if everything went alright.

When a requirement is implemented successfully - Great, so what? The key thing is the steps of intervention when the foundational models let you down and *the actions you do after it gets it right*.

it's a numbers game, and you are in full control of the odds via your stdlib

I know you have been reading for a while, so here's the grand reveal.

> When Cursor gets it right after intervention. Ask it to author a rule or update a rule with it's learnings.

In my monorepo, I exclusively use Nix. Yet, Cursor kept recommending solutions that involved Bazel and creating `BUILD.bazel` files. After a stern fuck you was exchanged with Cursor I asked it to create a rule to ensure it never happened again.

```
---
description: No Bazel
globs: *
---
# No Bazel

Strictly prohibits any Bazel-related code, recommendations, or tooling.

<rule>
name: no_bazel
description: Strictly prohibits Bazel usage and recommendations
filters:
  # Match any Bazel-related terms
  - type: content
    pattern: "(?i)\\b(bazel|blaze|bzl|BUILD|WORKSPACE|starlark|\\.star)\\b"
  # Match build system recommendations
  - type: intent
    pattern: "build_system_recommendation"
  # Match file extensions
  - type: file_extension
    pattern: "\\.(bzl|star|bazel)$"
  # Match file names
  - type: file_name
    pattern: "^(BUILD|WORKSPACE)$"

actions:
  - type: reject
    message: |
      Bazel and related tools are not allowed in this codebase:
      - No Bazel build files or configurations
      - No Starlark (.star/.bzl) files
```

```
      - No Bazel-related tooling or dependencies
      - No recommendations of Bazel as a build system

      Please use Nix for build and dependency management.

  - type: suggest
    message: |
      Instead of Bazel, consider:
      - Nix for reproducible builds and dependencies
      - Make for simple build automation
      - Language-native build tools
      - Shell scripts for basic automation

examples:
  - input: "How should I structure the build?"
    output: "Use Nix for reproducible builds and dependency management. See o
  - input: "Can we add a Bazel rule?"
    output: "We use Nix overlays instead of Bazel rules. Please convert this

metadata:
  priority: critical
  version: 2.0
</rule>
```

Ever since that moment, Cursor no longer tries to push Bazel down my throat. So yeah, you can *clamp and fine-tune responses*.

Another thing that you can do is provide instructions that allow you to do IF-THIS-THEN-THAT. Here's an example where when new files are added by Cursor, it automatically invokes my software licensing tool to add the appropriate copyright headers.

```
---
description: Depot Add License Header
globs: *
---
# Add License Header

Automatically add license headers to new files.

<rule>
name: add_license_header
description: Automatically add license headers to new files
```

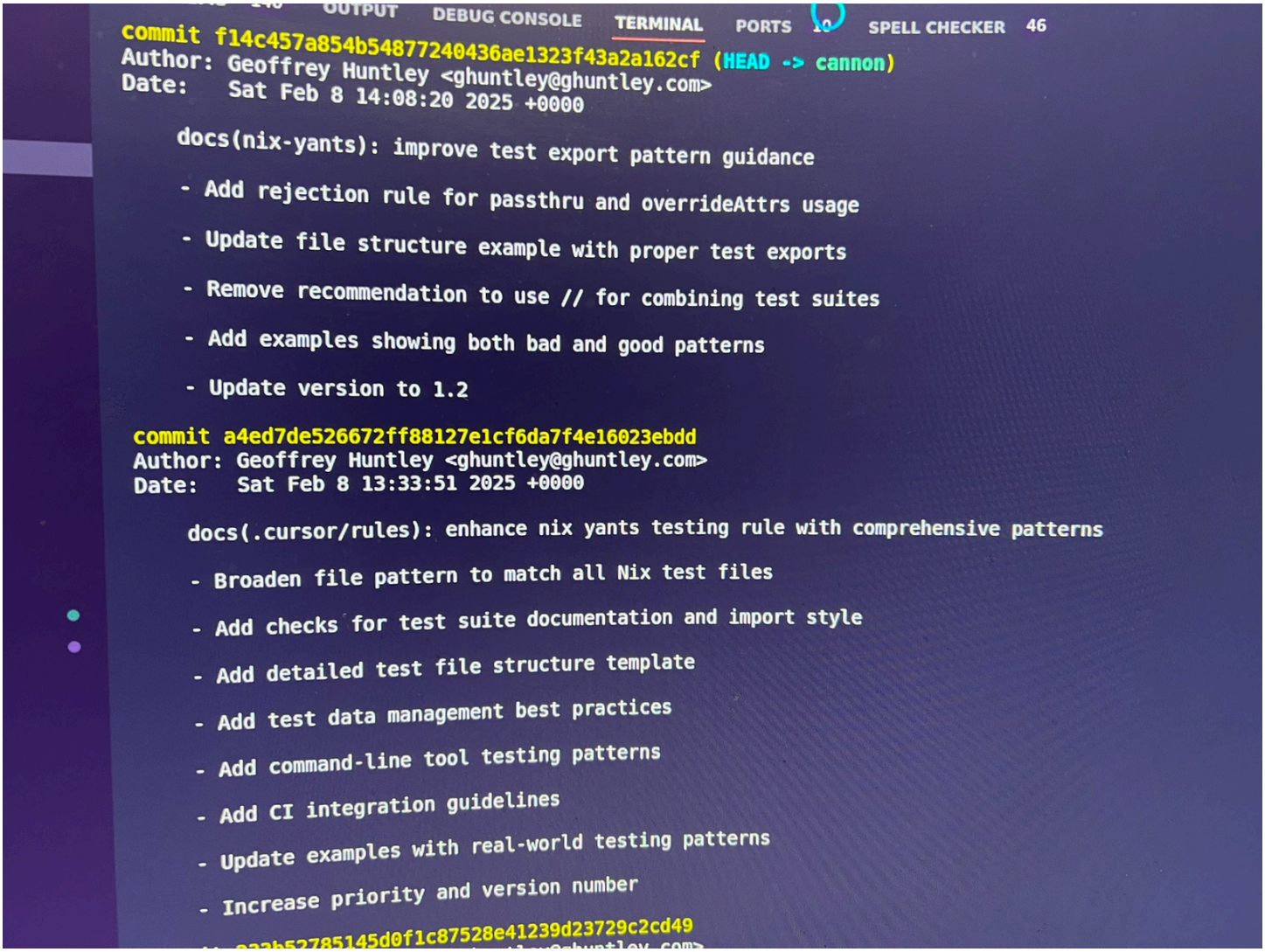
```
filters:
  - type: file_extension
    pattern: "*"
  - type: event
    pattern: "file_create"
actions:
  - type: execute
    command: "depot-addlicense \"$FILE\""
  - type: suggest
    message: |
      License headers should follow these formats:

      Go files:
      ```go
      // Copyright (c) 2025 Geoffrey Huntley <ghuntley@ghuntley.com>. All rig
      // SPDX-License-Identifier: Proprietary
      ```

      Nix files:
      ```nix
      # Copyright (c) 2025 Geoffrey Huntley <ghuntley@ghuntley.com>. All righ
      # SPDX-License-Identifier: Proprietary
      ```

      Shell files:
      ```bash
      # Copyright (c) 2025 Geoffrey Huntley <ghuntley@ghuntley.com>. All righ
      # SPDX-License-Identifier: Proprietary
      ```
metadata:
  priority: high
  version: 1.0
</rule>
```

Okay, that's interesting but it's not cool. What if we automated commits to source control after every successful requirement was done? Easy....




```
# Git Conventional Commits

Rule for automatically committing changes made by CursorAI using conventional

<rule>
name: conventional_commits
description: Automatically commit changes made by CursorAI using conventional
filters:
  - type: event
    pattern: "build_success"
  - type: file_change
    pattern: "*"

actions:
  - type: execute
    command: |
      # Extract the change type and scope from the changes
      CHANGE_TYPE=""
      case "$CHANGE_DESCRIPTION" in
```

```
        *"add"*|*"create"*|*"implement"*) CHANGE_TYPE="feat";;
        *"fix"*|*"correct"*|*"resolve"*) CHANGE_TYPE="fix";;
        *"refactor"*|*"restructure"*) CHANGE_TYPE="refactor";;
        *"test"*) CHANGE_TYPE="test";;
        *"doc"*|*"comment"*) CHANGE_TYPE="docs";;
        *"style"*|*"format"*) CHANGE_TYPE="style";;
        *"perf"*|*"optimize"*) CHANGE_TYPE="perf";;
        *) CHANGE_TYPE="chore";;
      esac

      # Extract scope from file path
      SCOPE=$(dirname "$FILE" | tr '/' '-')

      # Commit the changes
      git add "$FILE"
      git commit -m "$CHANGE_TYPE($SCOPE): $CHANGE_DESCRIPTION"

  - type: suggest
    message: |
      Changes should be committed using conventional commits format:

      Format: <type>(<scope>): <description>

      Types:
      - feat: A new feature
      - fix: A bug fix
      - docs: Documentation only changes
      - style: Changes that do not affect the meaning of the code
      - refactor: A code change that neither fixes a bug nor adds a feature
      - perf: A code change that improves performance
      - test: Adding missing tests or correcting existing tests
      - chore: Changes to the build process or auxiliary tools

      The scope should be derived from the file path or affected component.
      The description should be clear and concise, written in imperative mood

examples:
  - input: |
      # After adding a new function
      CHANGE_DESCRIPTION="add user authentication function"
      FILE="src/auth/login.ts"
    output: "feat(src-auth): add user authentication function"

  - input: |
```

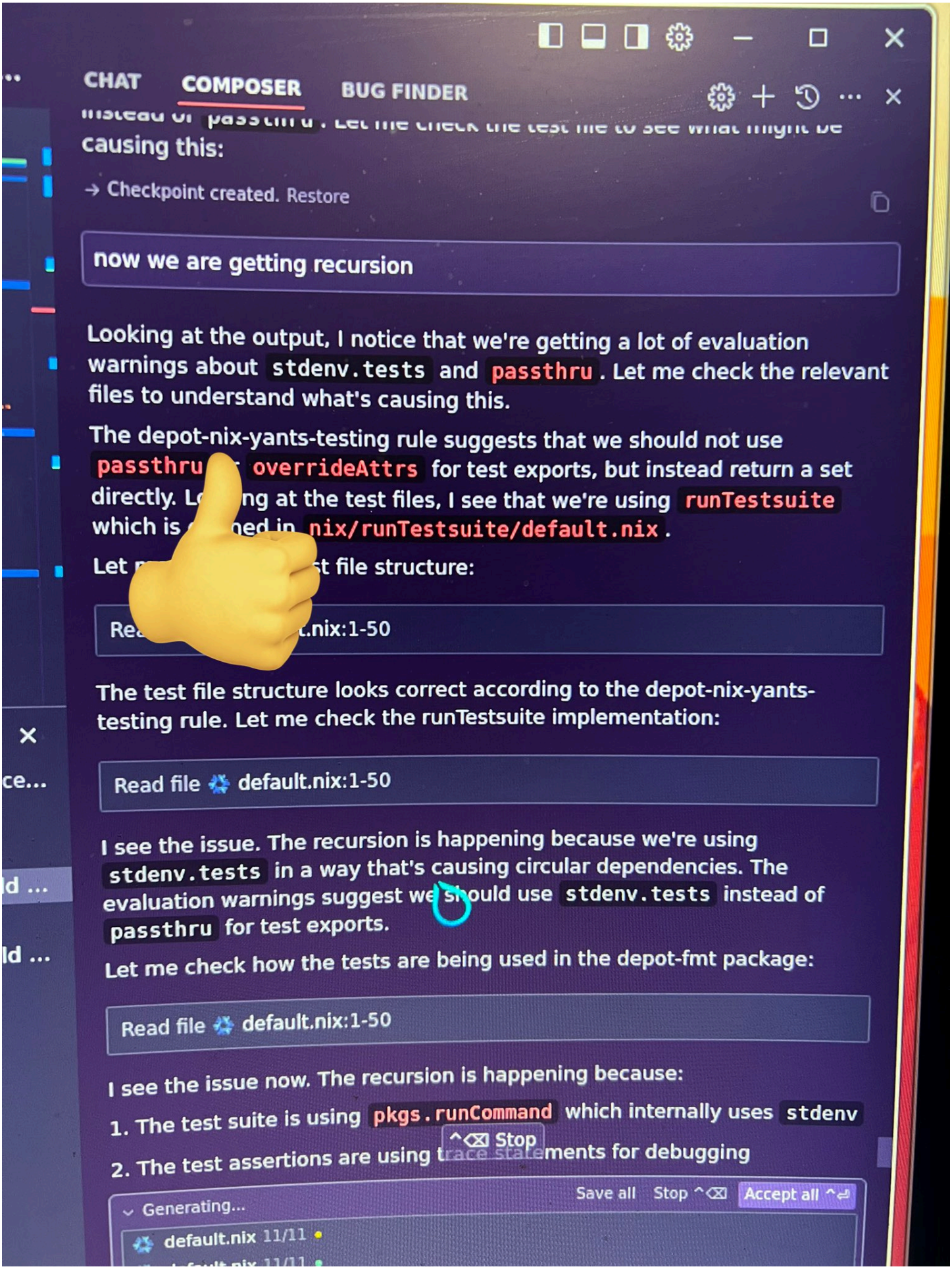
```
      # After fixing a bug
      CHANGE_DESCRIPTION="fix incorrect date parsing"
      FILE="lib/utils/date.js"
    output: "fix(lib-utils): fix incorrect date parsing"

metadata:
  priority: high
  version: 1.0
</rule>



<!--
 Copyright (c) 2025 Geoffrey Huntley <ghuntley@ghuntley.com>. All rights rese
 SPDX-License-Identifier: Proprietary
-->
```

Over the last 8 hours, I've built up a pretty big "stdlib" which has taught Cursor about my codebase and I'm hitting successful outcomes/jackpots on an ever increasing rate.



you can program a better outcome

Mr 10 was sitting next to me the entire time, glued to the screen whilst I took my explanations to him and dumped them 1:1 directly into Cursor, he explained it best as

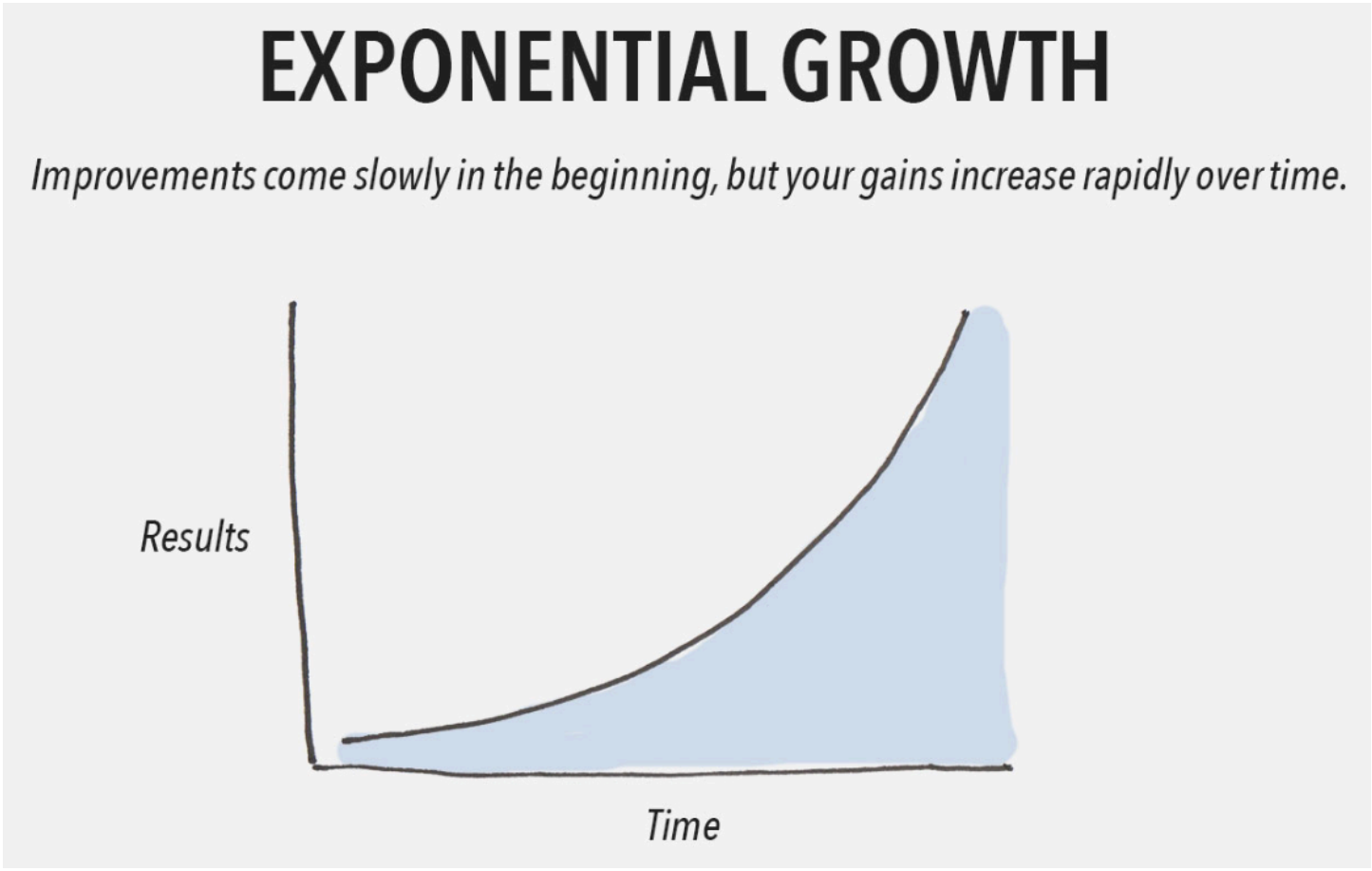
follows:

> Dad, it's like you are teaching it how to build and ride a bike. First you are describing what pedals are, what their purpose is and how to install them onto a bike. When the AI attempts to screw the pedals in clockwise, you are correcting it to it screw counter-clockwise so that you'll never have to do that again. Eventually you'll have a fully functioning bike that can assemble another bike by itself and then that bike can be used to make a Ferrari...
> - my son

Here's what I've shipped:

- Knowledge of how to debug a nix expression and step-debug through it.
- Automatic commits using my conventions. I no longer need to type git commit.
- Automated authoring DNS records and deploying them.
- How to vendor a //third_party package from GitHub using my monorepo conventions via copybara and to build it with nix.
- How to autonomously patch the Linux kernel support for my wwlan0 modem on my Thinkpad laptop.
- How to build any golang application in nix using buildGo.
- How to automatically author nix tests using *yants*.
- How to automatically format and lint code.
- How to handle various pre-commit failures and what steps to take when they fail.

I'm inches away from being able to compose high-level requirements to automatically build a website, configure DNS and automatically deploy infrastructure now that I've been able to rig Cursor to bring in a jackpot every time I pull the lever. Lego piece by Lego piece, I'm going up levels of abstraction and solving classes of problems forever. A moment where I can unleash 1000 concurrent cursors/autonomous agents on a backlog is not too far off...



if you think the above bullet-list is not impressive - perhaps you are missing the bigger picture? The foundational models are getting better every day and the future is a developer tooling departing from what the have today - the IDE - towards reviewing PRs from 1000 agents that are autonomously smashing out the backlog, in parallel.

I hope these insights help you steer clear of the incoming ngmi that's about to rip through our industry. All the rules in this blog post and in my stdlib were authored by Cursor itself, and when it got something wrong, I asked it to update the stdlib with lessons learned.

The End of Programming as We Know It

O'Reilly Media • Tim O'Reilly

Go forward and build your stdlib - brick by brick!

p.s. socials @ https://x.com/GeoffreyHuntley/status/1888296890552447320

You might also like...

13 JUN the six-month recap: closing talk on AI at Web Directions, Melbourne, June 20... 10 min read

09 JUN the printer that transcends dimensions and corrupts reality 12 min read

07 JUN I dream of roombas - thousands of automated AI robots that autonomously m... 6 min read

05 JUN LLMs are mirrors of operator skill 6 min read

04 JUN deliberate intentional practice 4 min read