Table of Contents

1. Code

1. GIT

1.1. git reset --hard

```
git log
```

```
git reset --hard a9b65fc3341da0dfb7d99aed912aafb09d709282
```

1.2. git restore --staged something

```
(No questions added yet)
```

2. GOAT Development Method

```
I've been coding with AI more or less since it became a thing,
and this is the first time I've actually found a workflow that
can scale across larger projects (though large is relative)
without turning into spaghetti. I thought I'd share since it may
be of use to a bunch of folks here.
```

```
Two disclaimers: First, this isn't the cheapest route--it makes
heavy use of Cline--but it is the best. And second, this really
only works well if you have some foundational programming
knowledge. If you find you have no idea why the model is doing
what it's doing and you're just letting it run amok, you'll have
a bad time no matter your method.
```

```
There are really just a few components:
```

```
A large context reasoning model for high-level planning (o1 or
gemini-exp-1206)
```

Cline (or roo cline) with sonnet 3.5 latest

A tool that can combine your code base into a single file

And here's the workflow:

1.) Tell the reasoning model what you want to build and collaborate with it until you have the tech stack and app structure sorted out. Make sure you understand the structure the model is proposing and how it can scale.

2.) Instruct the reasoning model to develop a comprehensive implementation plan, just to get the framework in place. This won't be the entire app (unless it's very small) but will be things like getting environment setup, models in place, databases created, perhaps important routes created as placeholders - stubs for the actual functionality. Tell the model you need a comprehensive plan you can "hand off to your developer" so they can hit the ground running. Tell the model to break it up into discrete phases (important).

3.) Open VS Code in your project directory. Create a new file called IMPLEMENTATION.md and paste in the plan from the reasoning model. Tell Cline to carefully review the plan and then proceed with the implementation, starting with Phase 1.

4.) Work with the model to implement Phase 1. Once it's done, tell Cline to create a PROGRESS.md file and update the file with its progress and to outline next steps (important).

5.) Go test the Phase 1 functionality and make sure it works, debug any issues you have with Cline.

6.) Create a new chat in Cline and tell it to review the implementation and progress markdown files and then proceed with Phase 2, since Phase 1 has already been completed.

7.) Rinse and repeat until the initial implementation is complete.

8.) Combine your code base into a single file (I created a simple Python script to do this). Go back to the reasoning model and decide which feature or component of the app you want to fully implement first. Then tell the model what you want to do and instruct it to examine your code base and return a

comprehensive plan (broken up into phases) that you can hand off to your developer for implementation, including code samples where appropriate. The paste in your code base and run it.

9.) Take the implementation plan and replace the contents of the implementation markdown file, also clear out the progress file. Instruct Cline to review the implementation plan then proceed with the first phase of the implementation.

10.) Once the phase is complete, have Cline update the progress file and then test. Rinse and repeat this process/loop with the reasoning model and Cline as needed.

The important component here is the full-context planning that is done by the reasoning model. Go back to the reasoning model and do this anytime you need something done that requires more scope than Cline can deal with, otherwise you'll end up with a inconsistent / spaghetti code base that'll collapse under its own weight at some point.

When you find your files are getting too long (longer than 300 lines), take the code back to the reasoning model and and instruct it to create a phased plan to refactor into shorter files. Then have Cline implement.

And that's pretty much it. Keep it simple and this can scale across projects that are up to 2M tokens--the context limit for gemini-exp-1206.

If you have questions about how to handle particular scenarios, just ask!

3. Code Library

3.1. Logo Maker James Fraze LLC

```
import svgwrite

import webbrowser
```

```
import os

font_size1 = 34

font_size2 = 22

font_size3 = 26

# Create the SVG canvas

dwg = svgwrite.Drawing("logo.svg", profile="tiny",
size=("500px", "500px"))

center = (250, 250) # Center of the canvas

# Define colors and font styles

colors = {

"background": "black",

"target_circle": "#242424", # Exact lighter gray for the
bullseye

"main_text": "#1da7f2",

"tagline_text": "#f2eb1d",

"sub_text": "white",

}

fonts = {

"main_text": "Comic Sans MS, cursive",

"tagline_text": "Comic Sans MS, cursive", # Simulates
handwritten style
```

```
"sub_text": "Comic Sans MS, cursive", # Helvitica-Bold

}

# Draw the background

dwg.add(dwg.rect(insert=(0, 0), size=("500px", "500px"),
fill=colors["background"]))

# Draw the bullseye rings as a single group

bullseye_group = dwg.g()

for i, radius in enumerate([90, 50, 10]):

bullseye_group.add(

dwg.circle(

center=center,

r=radius,

fill="none",

stroke=colors["target_circle"],

stroke_width=12 - i, # Thicker rings on the outside

)

)

dwg.add(bullseye_group)

# Add crosshair lines, ensuring no overlap at intersections

crosshair_group = dwg.g()
```

```
# Vertical crosshair
crosshair_group.add(
dwg.line(
start=(250, 100), # Top segment
end=(250, 210), # Before the ring
stroke=colors["target_circle"],
stroke_width=6,
)
)
crosshair_group.add(
dwg.line(
start=(250, 290), # After the ring
end=(250, 400), # Bottom segment
stroke=colors["target_circle"],
stroke_width=6,
)
)
# Horizontal crosshair
crosshair_group.add(
```

```
dwg.line(

start=(100, 250), # Left segment

end=(210, 250), # Before the ring

stroke=colors["target_circle"],

stroke_width=6,

)

)

crosshair_group.add(

dwg.line(

start=(290, 250), # After the ring

end=(400, 250), # Right segment

stroke=colors["target_circle"],

stroke_width=6,

)

)

dwg.add(crosshair_group)

# Calculate vertical centering for the tagline

baseline_shift = font_size2 / 3 # Approximate correction factor
for font baseline

# Add the main text ("James Fraze LLC")
```

```
dwg.add(

dwg.text(

"James Fraze llc",

insert=(250, 210),

fill=colors["main_text"],

font_family=fonts["main_text"],

font_size=f"{font_size1}px",

text_anchor="middle",

)

)

# Add the tagline ("Results Based") in a handwritten font,
perfectly centered

dwg.add(

dwg.text(

"Results Based",

insert=(250, 250 + baseline_shift), # Adjust for proper
centering

fill=colors["tagline_text"],

font_family=fonts["tagline_text"],

font_size=f"{font_size2}px",

text_anchor="middle",
```

```
)

)

# Add the subtext ("Digital Marketing")

dwg.add(

dwg.text(

"Digital Marketing",

insert=(250, 310),

fill=colors["sub_text"],

font_family=fonts["sub_text"],

font_size=f"{font_size3}px",

text_anchor="middle",

)

)

# Save the SVG file

svg_file_path = os.path.abspath("logo.svg")

dwg.save()

print(f"Logo saved as '{svg_file_path}'")

# Open the SVG file in the default web browser

webbrowser.open(f"file://{svg_file_path}")
```

```
That was a code block
```