THE CLAUDE GUIDE TO

Cloning Websites

A Field Manual for the AI Website Cloner Skill

Reconnaissance · Specification · Parallel Builders · Visual QA

“If a builder has to guess anything — a color, a font size, a padding value — you have failed at extraction.”

— the skill's first guiding principle

A Note on This Book

This book is a field manual for a specific, public piece of work: the AI Website Cloner skill, published by the developer JCodesMore as an open-source Claude Code template. It teaches you to point an AI coding agent at a live website and walk away with a real, buildable Next.js codebase — not a screenshot, not a guess, but a pixel-checked reconstruction with your own component files, your own git history, and your own room to customize.

The skill's own instructions call its author role a “foreman walking the job site.” That image runs through this whole book: someone who inspects one section at a time, writes down exactly what they find, and hands that writing to a specialist who builds it — while the foreman is already three sections further down the wall. Chapter 2 unpacks that model in full; every chapter after it assumes you understand why the skill refuses to build anything from a guess.

Two chapters are dedicated entirely to installation — one for Claude Code, where this skill was built to run, and one for Claude Cowork, for readers who want the guided, no-terminal experience or who are bringing a non-developer collaborator into the process. If you already know which surface you're using, you can skip to that chapter directly; the rest of the book applies to both.

The two case studies in Part III follow two invented practitioners — a freelance developer rebuilding a client's marketing site and an engineering lead migrating off a legacy stack whose original developer has left. Neither is a real person or a real company; they exist to make the pipeline concrete. Everywhere this book shows a spec file, a builder prompt, or a piece of output, treat it as a worked illustration of the pattern, not a claim about any particular site.

A closing note on scope: this skill reconstructs what is visible at a URL — layout, styling, interaction, content. It does not reconstruct a backend, and it is not a tool for impersonation, phishing, or passing off someone else's design as your own. Chapter 10 covers that boundary directly. Clone what you're entitled to rebuild: your own sites, sites you have permission to migrate, and sites you're studying to learn from — not to pass off as your own.

Contents

PART ONE

Foundations

What the skill is, how it thinks, and how to get it running

CHAPTER ONE

Why Clone a Website with AI

Priya Shah has quoted this job three times and lost it three times. A regional bakery chain wants their Squarespace site rebuilt as something they actually own — real code, real components, a repo they can hand to the next developer instead of a login page. Every agency she's competed against has quoted six weeks and a design-from-scratch process the client didn't ask for. The client doesn't want a redesign. They want their own site, the one they already like, in a codebase that isn't held hostage by a platform.

This is the job the AI Website Cloner skill was built for, and it's a more common job than it first sounds. A live website is a strange kind of artifact: fully specified — every pixel, every animation, every breakpoint is sitting right there in the browser — and yet, for most of the people who need the code behind it, completely inaccessible. The design exists. The build does not. Someone has to translate one into the other, by hand, one element at a time.

That translation used to take a frontend developer days of the most tedious work in the craft: open dev tools, note a font size, note a color, guess at a padding value, refresh, compare, guess again. The AI Website Cloner skill exists because that work is exactly the kind an AI coding agent is suited to — not because it's creative, but because it's mechanical, exhaustive, and unforgiving of shortcuts. A computer that can query getComputedStyle() on every element and never gets bored is better at this than a human who can.

What the skill actually does

Point Claude Code at one or more URLs, run the /clone-website command, and the skill drives a real browser through the page: full-page screenshots at desktop and mobile widths, a scroll from top to bottom watching for anything that moves on its own, a click on every button and tab, a hover over every card and link. It extracts the fonts, the color tokens, the favicons, every downloadable image and video. Then, section by section, it writes a specification file documenting exactly what it found — not an impression, but literal computed CSS values — and hands that file to a builder agent working in an isolated git branch. Builders work in parallel. Branches merge. The result is a Next.js codebase, built with shadcn/ui and Tailwind, that you can run, read, and modify like any project you wrote yourself.

The output is not a mockup and not a screenshot wrapped in an iframe. It's component files with real props, real breakpoints, and real interaction logic — the kind of codebase a second developer could open and immediately understand, because every section traces back to a spec file that says exactly why it looks the way it does.

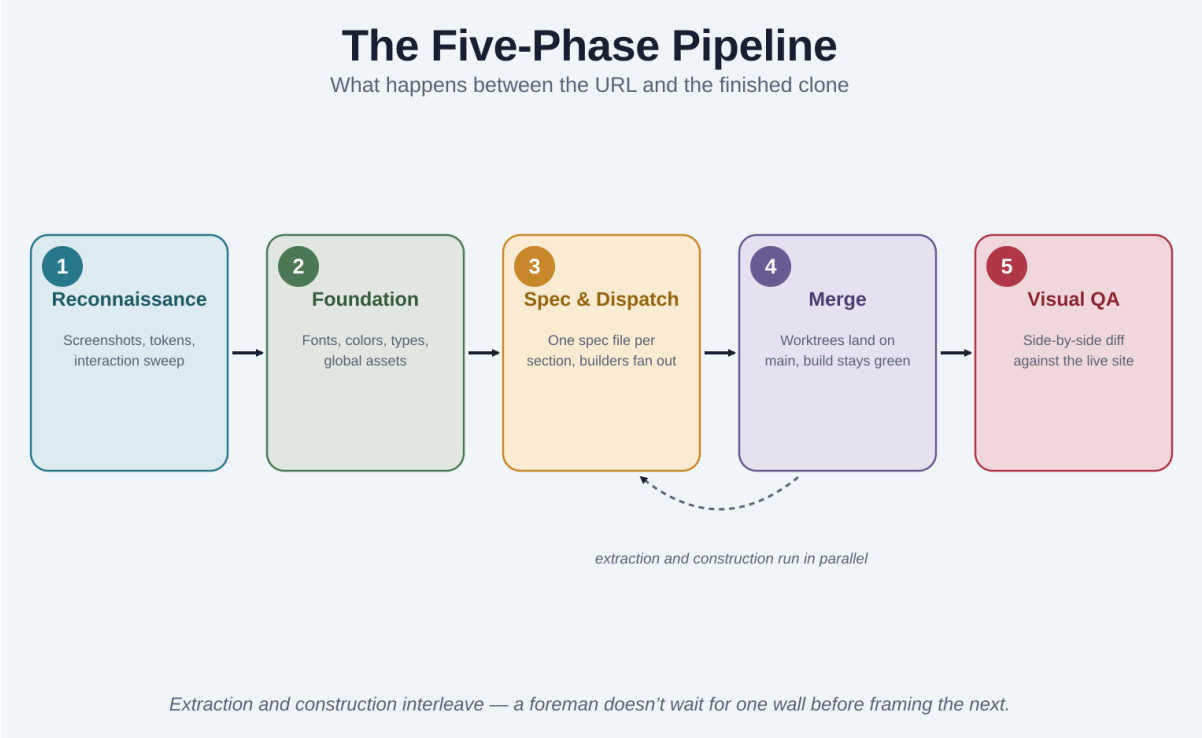

Figure 1. *The five phases the skill moves through for every clone, from first screenshot to final visual diff.*

Who reaches for this

Three situations come up again and again. The first is Priya's: a platform migration, where a site already exists and works, but the owner wants it out from behind a page builder and into a codebase they control. The second is site recovery — the original developer is gone, the repository was never handed over, and the only surviving copy of the design is the live site itself. The third is closer to Theo Lindqvist's use in later chapters: studying how a well-built site achieves a particular effect by rebuilding it, the way a musician learns a solo by ear.

What ties these together is that the goal is never to fool anyone. A phishing clone and a migration clone can look identical in a screenshot, which is exactly why the skill's own documentation is explicit about what it's for and what it categorically is not for — a boundary this book returns to directly in Chapter 10, and one worth internalizing before you run your first clone.

A live site is fully specified and completely inaccessible — the design exists; the build does not.

WATCH FOR

- Treating the tool as a design generator. It reconstructs what's on the page; it does not invent a better version of it.
- Skipping the terms-of-service check on a site you don't own before cloning it beyond personal study.
- Assuming a clone is a finished product — it's a pixel-accurate starting point, not a finished backend.

KEY TAKEAWAYS

- The skill turns a live URL into a real, editable Next.js codebase — not a mockup or a screenshot.
- It exists because CSS-by-hand extraction is mechanical, exhaustive work an agent can do more thoroughly than a person.
- Typical uses: platform migration, recovering a site whose source was lost, and studying production UI patterns.
- The output is pixel-checked against the original through a dedicated visual QA pass, covered in Chapter 7.
- What it builds is scoped to what's visible — layout, styling, interaction, and content, not backend systems.
- Using it to impersonate or misrepresent someone else's site is explicitly out of scope; Chapter 10 covers this in full.

EXERCISE

Try This

Self-Reflection

Think of a site you personally own or manage — a portfolio, a small business page, an old project.

Would you rather own its code today, or keep depending on whatever platform currently hosts it?

Collaborate

Before your next chapter, open that site in a browser and count how many distinct visual sections it has — hero, nav, feature grid, footer. You'll use that count in Chapter 5.

Reflection

What would you build differently if you owned the code instead of the page?

REFLECTION

- *Where in your own work has “the design already exists, but nobody has the code” shown up before?*
- *What's the difference between rebuilding a site you're migrating and rebuilding one you're studying?*

WHAT'S NEXT

Chapter 2 opens up the skill's own mental model — the “foreman” metaphor that explains why extraction and construction happen at the same time, and why every builder gets a written spec instead of a verbal summary.

CHAPTER TWO

Inside the Skill: The Foreman Model

Marcus Webb has watched two AI-assisted clone attempts fail before he ever heard of this skill. Both times, an agent looked at Fernwood Analytics' marketing site, took a few screenshots, and started writing component code straight from what it remembered seeing. Both times the result was close enough to pass a glance and wrong enough to fail a real comparison — a hover state missing here, a font weight guessed there, a whole animated section rebuilt as a static image because nobody checked whether it was a video.

The AI Website Cloner skill's documentation names this failure mode directly and gives it a fix: don't let an agent build from memory. The skill's own language for its process is architectural — it describes itself as a foreman walking a job site, not a two-phase pipeline of “look, then build.” A foreman doesn't inspect the whole building before any framing starts. They walk one wall, write down exactly what that wall needs, hand the note to a carpenter, and move on to the next wall while the carpenter works. Inspection and construction run at the same time, and nothing gets built from what the foreman merely remembers seeing an hour ago.

The five phases

Chapter 1 named the phases in passing; here is what each one is actually responsible for. Reconnaissance takes full-page screenshots at desktop and mobile, then runs a dedicated interaction sweep — scrolling slowly to catch anything that moves on its own, clicking every interactive element, hovering everything else — before extracting fonts, colors, and global patterns. Foundation is sequential and done by the orchestrator directly: global CSS, TypeScript types, icon extraction, and a real asset-download script, because nothing downstream can be built until the scaffold exists. Spec & Dispatch is the core loop, repeated once per page section: extract every computed style, write a spec file, then hand it to a builder working in its own git worktree. Merge folds each finished worktree back into the main branch, verifying the build stays green after every single merge. Visual QA closes the loop with a side-by-side comparison against the live site, at both viewport widths, testing every interaction — not just checking that the screenshot matches.

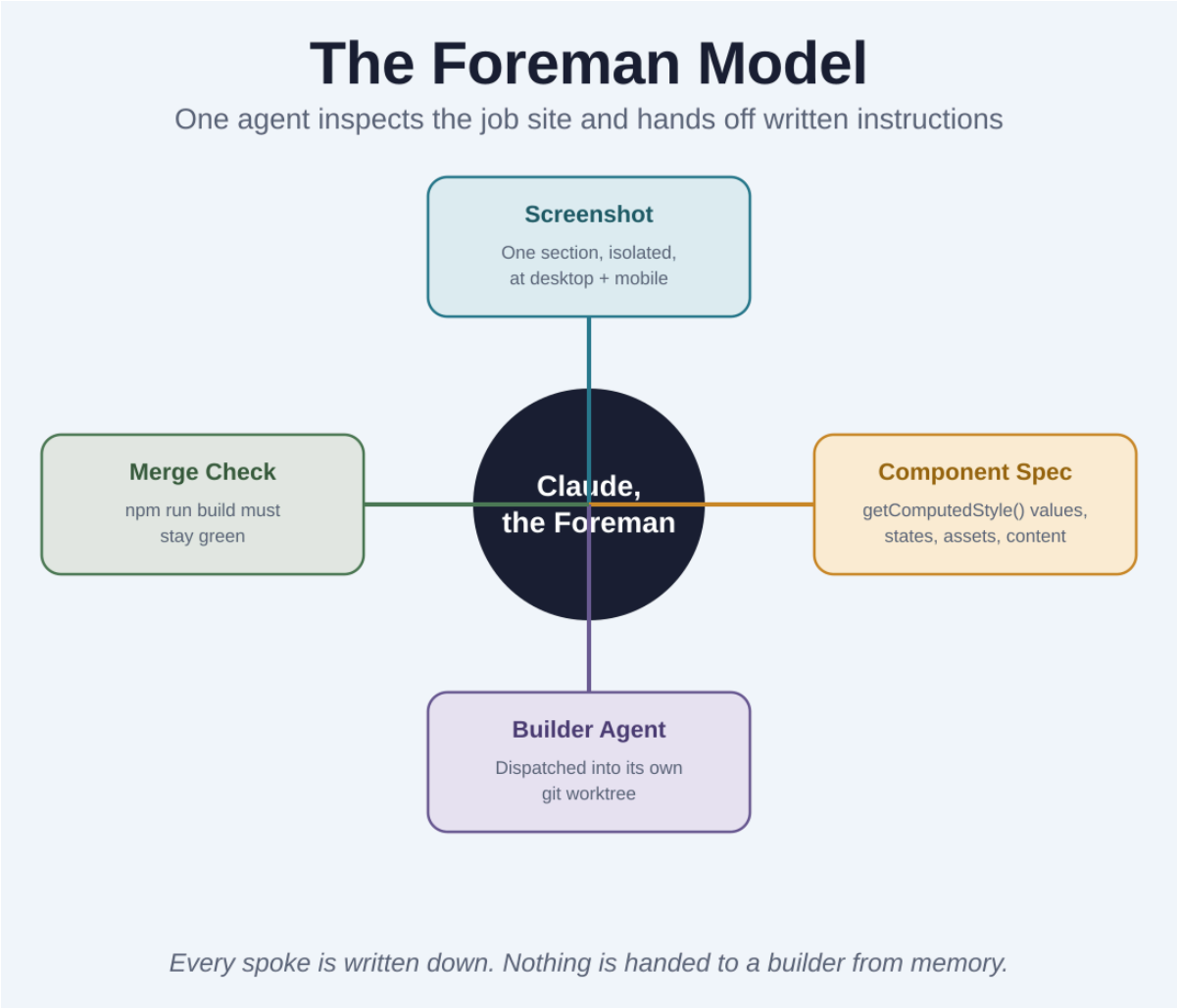

Figure 2. *The foreman's four core responsibilities: screenshot, spec, dispatch, and merge — always in that order, always written down.*

Nine principles worth memorizing

The skill's instructions distill down to nine non-negotiables, and they're worth having in your head before you run your first clone, because they explain almost every design decision in the chapters that follow. Completeness beats speed: if a builder has to guess a color or a padding value, extraction has failed, full stop. Small tasks, perfect results: an agent asked to build “the whole features section” approximates; an agent asked to build one card with exact values nails it — which is why Chapter 6 introduces a hard, mechanical line for when to split a section. Real content, real assets: this is a clone, not a mockup, so every image, video, and inline SVG gets pulled from the live site, layered overlays included. Foundation first: fonts, colors, and types are sequential and block everything else.

The next four concern behavior specifically, because a website is not a screenshot. Extract how it looks and how it behaves — the transition timing and trigger matter as much as the resting state. Identify the interaction model before building anything, because building a click-based tab bar when the original is

scroll-driven is a full rewrite, not a CSS tweak — Chapter 5 gives this its own decision tree. Extract every state, not just the default: a tab bar showing “Featured” on load still has three other tabs with their own content waiting to be clicked. And spec files are the source of truth — a builder never receives a summary from the foreman's memory, only the full, written contents of its own spec file. The ninth principle is the simplest and the least negotiable: the build must always compile. Every builder verifies its own work before finishing; every merge is checked again.

THE METHOD

The Foreman's Loop, one section at a time

1. Screenshot the section in isolation, at desktop and mobile widths.
2. Extract every computed style with a scripted DOM walk — never hand-measured.
3. Determine the interaction model before writing a single spec line: static, scroll, click, hover, or time-driven.
4. Write the full spec file to disk — DOM structure, styles, states, assets, responsive behavior.
5. Dispatch a builder (or several, if the section is complex) into its own git worktree with that spec inline.
6. Move immediately to the next section — don't wait for the builder to finish.
7. As worktrees complete, merge them and re-verify the build.

A foreman doesn't inspect the whole building before any framing starts — they walk one wall at a time.

WHY IT'S NOT TWO PHASES

A strict “inspect everything, then build everything” pipeline sounds safer, but it means the agent is reconstructing dozens of sections from stale memory by the time it reaches the last one. Interleaving keeps every build decision one spec-read away from the extraction that produced it.

KEY TAKEAWAYS

- The skill models itself as a foreman: write down what you find, hand it off, keep moving — never build from memory.
- Five phases run per clone: Reconnaissance, Foundation, Spec & Dispatch, Merge, and Visual QA.
- Extraction and construction interleave section by section rather than running as two strict, sequential phases.
- Nine guiding principles anchor every decision in this book, from spec files to the complexity budget to build gating.
- “Completeness beats speed” and “the build must always compile” are the two principles every other one supports.

- Foundation — fonts, colors, types, global assets — is sequential and blocks all section-level work.

EXERCISE

Try This

Self-Reflection

Recall a time you (or a tool) built something from a half-remembered description instead of the real spec. What broke?

Collaborate

Open a site you know well. Pick one section and try to write its complete CSS spec from memory before looking. Then check dev tools — how many values did you get wrong?

Reflection

Which of the nine principles would have prevented the mistake you just made in the exercise above?

REFLECTION

- *Why does "write it down" matter more for an AI agent than it might for a human developer working alone?*
- *Where else in your own workflow would a written spec, handed to a specialist, outperform a verbal summary?*

WHAT'S NEXT

With the model in place, Chapters 3 and 4 get the skill running — first in Claude Code, where it was built to live, then in Claude Cowork, for a guided, no-terminal path.

CHAPTER THREE

Installing the Skill in Claude Code

BY THE END OF THIS CHAPTER YOU'LL BE ABLE TO

✓ Create your own working copy of the AI Website Cloner template, correctly, without opening a pull request back to the original.

✓ Get browser automation connected so the skill can actually drive a page.

✓ Run your first /clone-website command and know what success looks like.

Priya opens the repository on GitHub for the first time expecting a library she'll npm-install into her existing project. It isn't one. The AI Website Cloner is a template repository — a full, pre-scaffolded Next.js codebase with the skill already wired into it — and the very first thing its own README insists on is that you not clone it directly. You start your own copy, on your own account, and the clone happens inside that copy.

Why it's a template, not a package

This matters more than it looks like it should. The skill's builder agents assume a specific, already-working foundation: Next.js 16 with the App Router, React 19, TypeScript in strict mode, shadcn/ui components on Tailwind CSS v4, and a project structure with public/images, public/videos, and docs/research already in place. A skill you dropped into an unrelated project would have nothing to build onto — no design tokens to overwrite, no shadcn primitives to import, no icons.tsx to populate. Starting from the template isn't a formality; it's the foundation phase from Chapter 2, done once, before you ever run the skill.

THE METHOD

Getting a Working Clone Skill, Step by Step

1. On the template's GitHub page, click “Use this template,” then “Create a new repository.” Name it, choose public or private, and leave “Include all branches” off. This gives you a separate repository under your own account — your changes stay yours.
2. Clone your new repository locally, not the original template: git clone https://github.com/YOUR-USERNAME/YOUR-NEW-REPOSITORY.git, then cd into it.
3. Install dependencies: npm install. This pulls in Next.js, shadcn/ui, Tailwind, and the docx/PDF/office tooling the skill's later phases lean on.
4. Confirm Node.js 24 or later is active — the template's own prerequisites list this as a hard requirement.
5. Connect browser automation. The skill's pre-flight check refuses to run without it: Chrome MCP, Playwright MCP, Browserbase MCP, or Puppeteer MCP all work, and Chrome MCP is what the template

recommends. Install the Chrome MCP extension from the Chrome Web Store, then launch Claude Code with it enabled: claude --chrome.

6. From inside that Claude Code session, run the skill itself: /clone-website <target-url1> [<target-url2> ...]. For a first run, use a single URL — your own site if you have one, so there's nothing to ask permission for.

```
claude --chrome
/clone-website https://your-own-site.example
```

What a healthy first run looks like

You should see the terminal fill with a foreman's running commentary, not silence: a note that it's checking for browser MCP tools, screenshots being taken and saved to docs/design-references/, a pass described as an “interaction sweep,” then component spec files appearing one at a time in docs/research/components/ as builder agents get dispatched. If a build step fails partway through, that's expected and not a sign anything is broken — the ninth guiding principle from Chapter 2 means the skill checks npm run build after every merge and stops to fix problems rather than plowing through them.

Priya's first run is against her own single-page portfolio, deliberately small. It finishes in under fifteen minutes, with four component spec files sitting in docs/research/components/ and a page.tsx she can open in her editor and recognize instantly — a HeroSection, a WorkGrid, a ContactForm, a Footer, each traceable to a spec file that explains exactly why it looks the way it does. That legibility is the whole point: the next developer who opens this repo, whether that's Priya in six months or someone she hands the project to, isn't reverse-engineering a black box.

If you already have a project

Sometimes you don't want a brand-new repository — you want the skill available inside a project you're already building. This is possible but not casual: copy .claude/skills/clone-website/ from the template into your own project's .claude/skills/ directory, and make sure your project already matches the assumptions the builder prompts carry — Next.js App Router, shadcn/ui, Tailwind v4, and the same public/ and docs/research/ structure. If your stack diverges, expect to adapt the spec-file template and the builder instructions in Phase 3 of the skill yourself; this is the advanced path, not the default one.

Starting from the template isn't a formality — it's the foundation phase, done once, before you ever run the skill.

WATCH FOR

- Cloning the original template repository directly instead of your own “Use this template” copy — your changes have nowhere safe to live.
- Skipping the Chrome MCP extension install and being surprised when the skill asks which browser tool you have.
- Running the skill against a large, multi-page site as your very first test. Start with one page.

KEY TAKEAWAYS

- The skill ships inside a template repository, not as a standalone package — use GitHub's “Use this template,” never clone the original directly.
- npm install and Node.js 24+ get the pre-scaffolded Next.js + shadcn/ui + Tailwind base ready to build onto.
- Browser automation is mandatory; Chrome MCP is the recommended choice and claude --chrome launches Claude Code with it enabled.
- The skill itself runs as /clone-website <url> from inside a Claude Code session.
- A healthy first run narrates itself: screenshots, an interaction sweep, spec files appearing, builders dispatching.
- You can add the skill to an existing project by copying .claude/skills/clone-website/, but your project must already match its structural assumptions.

EXERCISE

Try This

Self-Reflection

Do you have a small site — a portfolio, a landing page, a side project — you could safely use as a first test?

Collaborate

- Create your own copy of the template via “Use this template.”
- Run npm install and confirm the base project builds with npm run build before you touch the skill.
- Connect Chrome MCP and run /clone-website against your chosen test site.

Reflection

What in the terminal output told you the skill was following the foreman model from Chapter 2, rather than just generating code?

REFLECTION

- *Why might a skill this specific refuse to work well once you strip away the scaffold it was built against?*
- *What would you need to change in your own projects to make them a safe home for this skill?*

WHAT'S NEXT

> *Not every collaborator wants a terminal. Chapter 4 covers the same skill from inside Claude Cowork — what transfers cleanly, and what the guided surface can't do that Claude Code can.*

CHAPTER FOUR

Installing and Running the Skill in Claude Cowork

BY THE END OF THIS CHAPTER YOU'LL BE ABLE TO

✓ Add the clone-website skill to Claude Cowork and enable it for a session.

✓ Understand honestly which parts of the pipeline Cowork can run today, and which still need Claude Code.

✓ Set up a handoff pattern that lets a non-developer collaborator drive reconnaissance while an engineer runs the full build.

Dana Okafor runs design at Northlight Studio, and she is not a developer. She can read a Figma file fluently and she knows exactly what's wrong with the marketing site their last contractor left behind, but she has never opened a terminal on purpose. When her engineering lead mentions the AI Website Cloner skill, her first question isn't “how do I run this,” it's “can I even open it.”

What Cowork is built for

Claude Cowork is Anthropic's guided, desktop-based agentic app for people who want an AI agent to handle multi-step work without living in a terminal. Skills work inside it: from Customize in the sidebar, opening the directory and selecting the Skills tab shows every skill available to install, and skills you enable there apply automatically — you type / to see them explicitly, or simply describe the task and Cowork recognizes when a skill applies. Code execution has to be turned on first, in Settings under Capabilities, since skills depend on it to run.

The clone-website skill isn't in Anthropic's public skill directory, because it ships bundled inside its own GitHub template rather than as a standalone submission. That means getting it into Cowork is a custom-skill upload, not a directory browse: take the SKILL.md from .claude/skills/clone-website/ in the template, package it as its own skill folder, and add it to your account from Customize > Skills the same way you'd add any personal skill — reviewing its contents first, which is good practice for any skill from outside the official directory, and doubly worth doing for one that instructs an agent to control a browser.

Figure 3. *The same pipeline, two surfaces. Claude Code drives the full build; Claude Cowork is best scoped to guided, single-page work.*

Being honest about the gap

Here is the part worth saying plainly rather than glossing over: this particular skill leans hard on things Claude Code provides natively that a guided desktop surface does not — full filesystem and terminal access, git worktrees that let multiple builder agents work in parallel without stepping on each other, and a persistent Next.js project that survives across a long session. Cowork's execution model is intentionally more contained. That doesn't make it the wrong tool; it makes it a different-scoped one.

What transfers cleanly is the front half of the pipeline — reconnaissance and spec-writing, the reading-and-documenting work that doesn't require parallel git branches. Dana can open Cowork, point it at Northlight's own marketing site, and have it produce the screenshots, the design-token summary, and a page-topology document exactly as described in Chapter 5, all without touching a repository. What doesn't transfer as cleanly is Phase 3's dispatch-to-worktree pattern and the merge step in Phase 4 — those assume Claude Code's environment. For a single, simple page, Cowork can carry a build through as one continuous session rather than parallel branches; for anything with more than a couple of complex sections, the complexity budget from Chapter 6 argues for handing that spec pack to Claude Code.

THE METHOD

The Handoff Pattern

1. In Cowork, run reconnaissance against the target: screenshots, the design-token extraction, the interaction sweep. Save everything as files — this is exactly what Chapter 5 calls the BEHAVIORS.md and PAGE_TOPOLOGY.md artifacts.
2. For a small, single-section rebuild, let Cowork carry the whole thing through in one guided session — it can write and apply straightforward component code without needing parallel worktrees.
3. For anything larger, export the spec files and hand them to a teammate running Claude Code, who dispatches builders and merges in parallel exactly as Chapter 6 and Chapter 7 describe.
4. Either way, the spec file is still the source of truth — principle eight from Chapter 2 doesn't change because the surface did.

The spec file is still the source of truth — that principle doesn't change because the surface did.

WHERE COWORK GENUINELY WINS

Bringing a non-developer collaborator into the loop early. Dana can review a component spec in plain language, catch “that's not our real tagline” before a builder ever runs, and hand a cleaner brief to engineering than a screenshot and a Slack message ever gave them.

WATCH FOR

- Expecting Cowork to run the exact same parallel git-worktree pipeline Claude Code does — it's a different execution model, not a smaller terminal.
- Uploading a skill from outside Anthropic's directory without reading what it does first, especially one that drives a browser.
- Forgetting to enable code execution in Settings > Capabilities — skills silently can't run without it.

KEY TAKEAWAYS

- Cowork installs skills from Customize > Skills; code execution must be enabled first in Settings > Capabilities.
- Because clone-website isn't in the public skill directory, adding it to Cowork means uploading it as a personal custom skill.
- Reconnaissance and spec-writing — the front half of the pipeline — run well in Cowork's guided, no-terminal environment.
- Parallel git worktrees and multi-agent dispatch are Claude Code's strength; Cowork is better scoped to small, single-session rebuilds.

- A practical pattern: run reconnaissance in Cowork, then hand the spec files to Claude Code for anything beyond a simple page.
- Cowork's real advantage is bringing non-developer collaborators into spec review before a single builder agent runs.

EXERCISE

Try This

Self-Reflection

Think about who on your team would benefit from reviewing a plain-language component spec before code gets written. Who is that, for you?

Collaborate

Enable code execution in Cowork's settings, then add the clone-website skill as a custom skill and confirm it appears when you type /.

Run reconnaissance only — screenshots and token extraction — against a small page, and review the output before deciding whether to build further.

Reflection

Where does your own handoff between design and engineering break down today, and would a written spec file have prevented it?

REFLECTION

- *What does Dana gain by reviewing a spec file in Cowork that she wouldn't get from a screenshot and a verbal description?*
- *When would you deliberately choose the more limited surface over the more powerful one?*

WHAT'S NEXT

Part II moves from setup to execution. Chapter 5 opens the pipeline itself, starting with the phase every clone lives or dies on: reconnaissance.

PART TWO

Running the Pipeline

Reconnaissance, specification, parallel builders, and the QA pass that closes the gap

CHAPTER FIVE

Reconnaissance and the Interaction Sweep

BY THE END OF THIS CHAPTER YOU'LL BE ABLE TO

✓ Run a complete reconnaissance pass: screenshots, global tokens, and the three-part interaction sweep.

✓ Recognize why scroll-driven and click-driven interfaces must be tested in that exact order.

✓ Produce the two artifacts — BEHAVIORS.md and PAGE_TOPOLOGY.md — every later phase depends on.

Theo Lindqvist has rebuilt three competitor landing pages this year, purely to understand how they work. His second attempt taught him the lesson this chapter is built around: he rebuilt a pricing section's tab bar as click-to-switch, because that's what it looked like from a screenshot, and it was wrong. The real site swapped tabs automatically as you scrolled past each plan, using an IntersectionObserver he never noticed because he clicked before he scrolled.

Screenshots and global extraction

Reconnaissance opens with full-page screenshots at two widths — 1440px for desktop, 390px for mobile — saved to docs/design-references/ as the master reference every later spec file will point back to. Before touching any individual section, the skill extracts what applies to the whole page: every font family and weight actually in use, the full color palette mapped against shadcn's token names, favicons and metadata, and any sitewide pattern like a smooth-scroll library or a global keyframe animation. This is Chapter 2's “foundation first” principle in practice — fonts and colors get locked in before any section-level work begins, because every builder downstream will import from that same foundation.

The interaction sweep, in order

Then comes the pass that catches what a screenshot never will. It runs in three parts, and the order is not arbitrary. First, a scroll sweep: move down the page slowly and watch, without clicking anything, for a header that changes appearance, elements that animate into view, a sidebar or tab indicator that switches on its own, scroll-snap points, or a smooth-scroll library changing how the page feels to move through. Second, a click sweep: now click every button, tab, pill, link, and card, and for anything with multiple states — tabs especially — click each one individually and record what appears. Third, a hover sweep across every card, link, image, and nav item, noting exactly what changes and how it transitions.

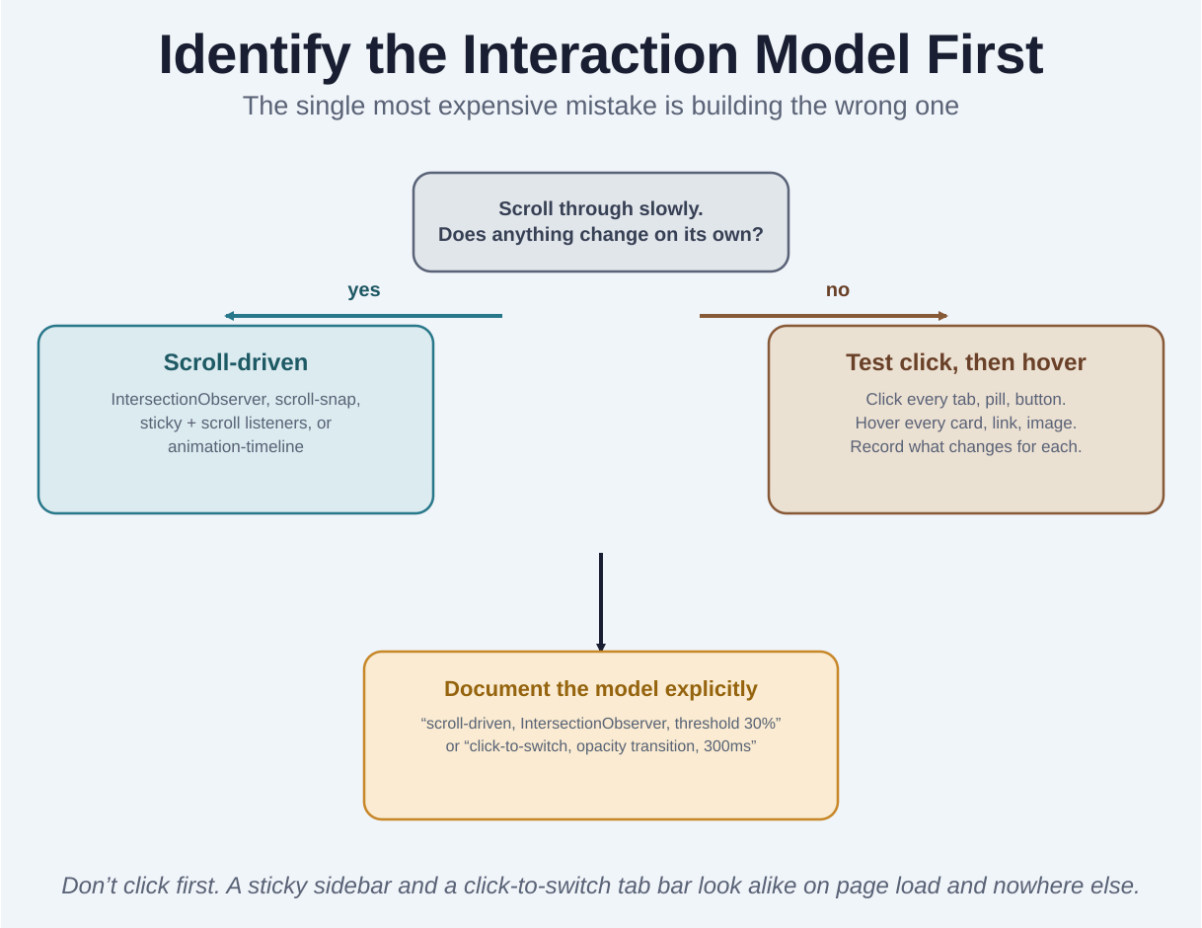

Figure 4. *Scroll first, always. Clicking before you've ruled out scroll-driven behavior is how a sticky sidebar gets rebuilt as a tab bar.*

The reason for that order is Theo's mistake, generalized: a scroll-driven interface and a click-driven one can look identical on first page load, and the only way to tell them apart is to scroll first and watch for anything moving on its own before you ever touch a button. Getting this backwards doesn't cost you a CSS tweak later — it costs a full rewrite of the section's interaction logic, because the two patterns are implemented in completely different ways: an IntersectionObserver watching scroll position versus an onClick handler swapping visible content.

A responsive sweep closes reconnaissance out: the same page tested at 1440px, 768px, and 390px, noting exactly where a layout changes — a sidebar disappearing, columns collapsing to a stack — and the approximate pixel width where it happens. Skipping this is one of the most common ways a clone looks perfect on a laptop and falls apart on a phone.

Two documents, not just memory

Everything from the sweep gets written to docs/research/BEHAVIORS.md — not summarized, not held in context, written to disk as a reference every component spec in Chapter 6 will cite. Alongside it, a page-topology pass maps every distinct section top to bottom, naming each one, noting which are sticky

overlays versus normal flow content, and recording each section's interaction model in one line, so that by the time extraction moves to Chapter 6's per-section specs, the scroll-versus-click question has already been answered for every part of the page.

THE METHOD

The Sweep, Section by Section

1. Screenshot the full page at 1440px and 390px before anything else.
2. Extract fonts, colors, favicons, and any sitewide scroll or animation library.
3. Scroll slowly, top to bottom, without clicking. Note anything that changes on its own.
4. Click every interactive element. For anything with multiple states, click each state and record its content.
5. Hover every card, link, image, and nav item. Note the transition, not just the end state.
6. Test the same page at 1440px, 768px, and 390px, and note where the layout changes.
7. Write it all to BEHAVIORS.md and PAGE_TOPOLOGY.md before moving to Phase 3.

Don't click first. A sticky sidebar and a click-to-switch tab bar look identical on page load and nowhere else.

WATCH FOR

- Clicking before scrolling — the single most expensive ordering mistake in the whole pipeline.
- Testing responsiveness only at desktop width and discovering the mobile layout breaks after the clone is already built.
- Treating the interaction sweep as optional for a page that “looks static” — static-looking pages still have hover states worth capturing.

KEY TAKEAWAYS

- Reconnaissance opens with full-page screenshots at 1440px and 390px — the master reference for every later spec.
- Global extraction — fonts, colors, favicons, sitewide patterns — happens before any per-section work, per the foundation-first principle.
- The interaction sweep runs scroll, then click, then hover, in that specific order, because scrolling reveals what clicking can hide.
- Every stateful element — tabs especially — needs each state clicked and recorded individually, not just the default view.
- A responsive sweep at three widths catches layout breakpoints before they become a rebuild.

- BEHAVIORS.md and PAGE_TOPOLOGY.md are written artifacts, not working memory — every later spec file cites them.

EXERCISE

Try This

Self-Reflection

Recall the site you counted sections on back in Chapter 1. Which of its sections do you already suspect are scroll-driven rather than click-driven?

Collaborate

- Scroll through that site slowly, without clicking anything, and list what changes on its own.
- Now click through it and list what changes only on click.
- Compare the two lists — anything that surprised you either way?

Reflection

What would have gone wrong if you'd clicked first?

REFLECTION

- *Why does testing order — scroll, then click, then hover — matter more than testing thoroughness alone?*
- *What's the cost difference between fixing a wrong interaction model early versus after a builder has already coded it?*

WHAT'S NEXT

Chapter 6 turns everything reconnaissance found into the artifact that actually gets handed to a builder: the component spec file, and the rule for when one section needs to become several.

CHAPTER SIX

Specs, Dispatch, and the Builder Army

BY THE END OF THIS CHAPTER YOU'LL BE ABLE TO

- ✓ Write a component spec file with every section the skill's template requires.
- ✓ Apply the ~150-line complexity budget to decide when a section needs more than one builder.
- ✓ Understand exactly what a builder agent receives — and why it's never told to "go read the spec file."

Theo's third attempt at that pricing page went differently. This time, before writing a single component, he wrote six spec files — one for the section wrapper and one for each of the three pricing cards, plus two for a comparison table underneath that turned out to have its own distinct hover behavior. The pricing section alone would have blown past 150 lines of spec content as one file. Split six ways, each builder got a focused, perfectly specified brief instead of an overwhelming one.

Anatomy of a spec file

Every component gets a file at docs/research/components/<component-name>.spec.md, written before any builder is dispatched, and it follows the same six-part shape every time. An Overview names the target file, the reference screenshot, and the interaction model determined in Chapter 5. DOM Structure describes the element hierarchy — what contains what. Computed Styles lists exact getComputedStyle() values for the container and every meaningful child — never an estimate, never "around 16px." States & Behaviors documents every trigger, the before-and-after values, and the transition timing for anything that changes. Assets & Content lists local asset paths and verbatim text, copied from the live site rather than paraphrased. Responsive Behavior closes it out with what changes at desktop, tablet, and mobile, and roughly where the breakpoint sits.

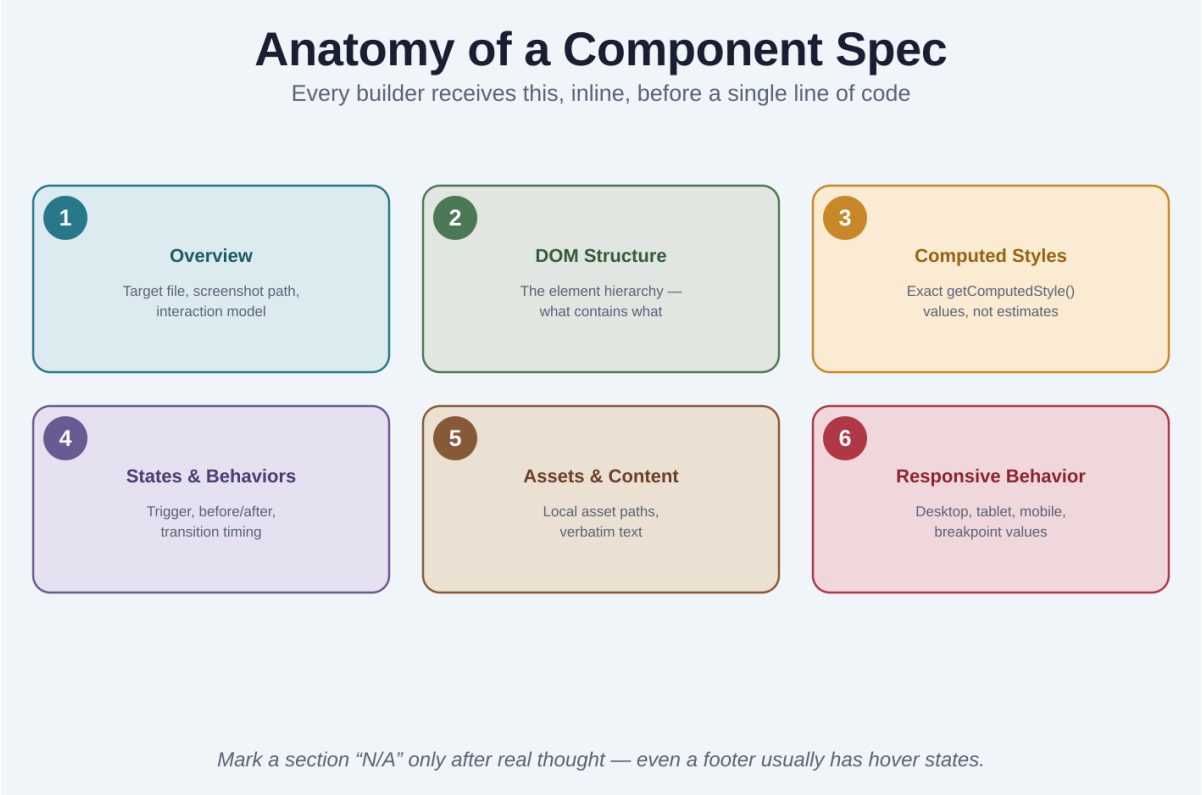

Figure 5. *Six sections, every time. A section without a documented state still gets "N/A" written deliberately — never left blank.*

The instruction to mark an inapplicable section "N/A" rather than skip it is more important than it looks. A footer with no visible animation still usually has link hover states; leaving States & Behaviors blank instead of writing "N/A" after actually checking is how a real hover effect quietly goes unbuilt.

The complexity budget

Not every section gets one builder. The rule is mechanical, not a judgment call: if a builder prompt would exceed roughly 150 lines of spec content, the section needs to be split, full stop — not overridden because "it's all related." A simple banner with a heading and a button is one small spec and one builder. A features section with three card variants, each with its own hover states and internal layout, is one spec and one builder per card variant, plus a fourth for the wrapper that imports them.

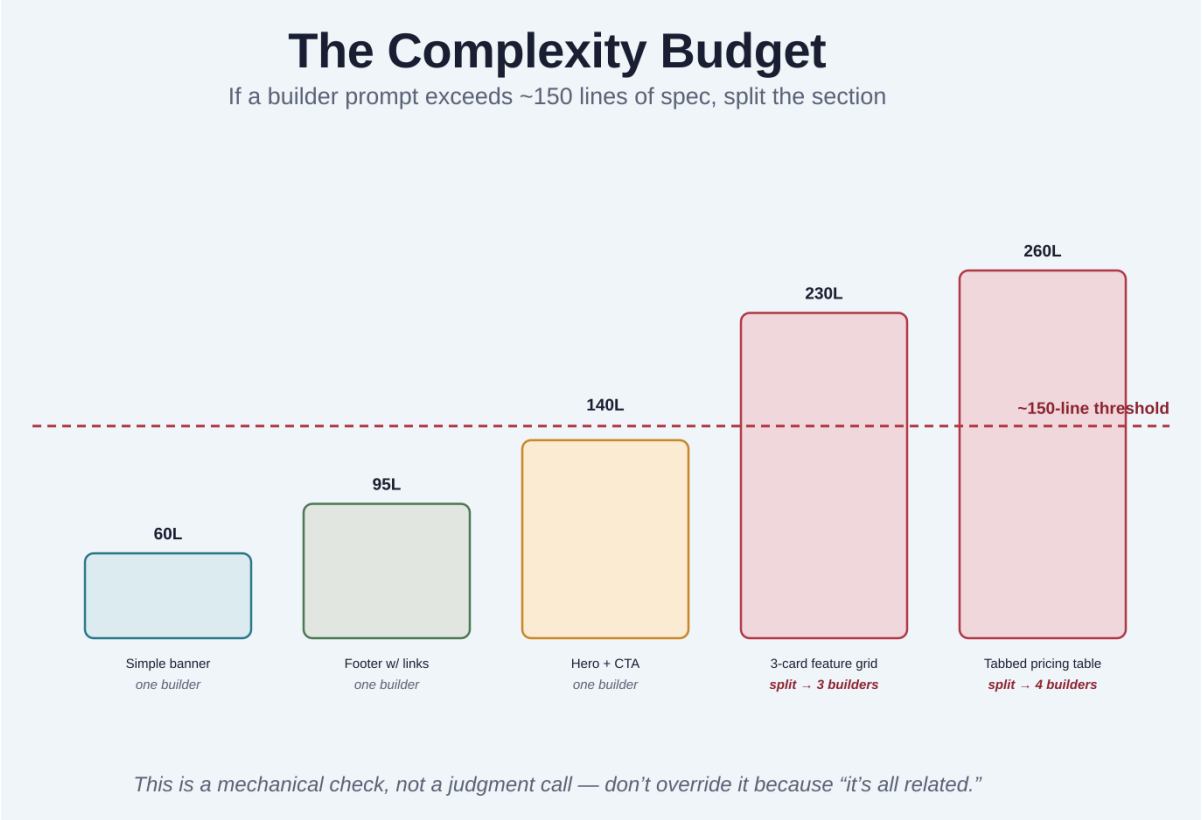

Figure 6. *Five sections against the same threshold. The two that cross it get split — not because the rule prefers it, but because it's mechanical.*

This connects directly to "small tasks, perfect results" from Chapter 2: an agent asked to build one card with exact values nails it every time; an agent asked to build an entire complex section approximates. Splitting isn't extra caution — it's the difference between a builder that has room to get every value exactly right and one that's silently trading precision for scope.

What a builder actually receives

Every builder agent gets the same package, and none of it is optional: the full contents of its component spec file, pasted inline into the prompt — never a pointer to go read the file itself; the path to its reference screenshot; which shared resources to import, like icons.tsx or shadcn primitives; the exact target file path it should create; and an instruction to run npx tsc --noEmit before declaring itself finished. As soon as one section's builders are dispatched, the foreman moves straight to extracting the next section rather than waiting — builders work in their own git worktrees in parallel while extraction continues.

> THE METHOD
>
> ### Dispatch Checklist, Before Any Builder Runs
>
> 1. Confirm every CSS value in the spec came from getComputedStyle(), not an estimate.

2. Confirm the interaction model is documented explicitly — static, scroll, click, hover, or time-driven.
3. For stateful components, confirm every state's content and styles were captured, not just the default.
4. Confirm all images are identified, including layered overlays and background compositions.
5. Confirm the spec is under ~150 lines of content — if not, split the section before dispatching anything.
6. Confirm text content is verbatim, not paraphrased.

An agent asked to build one card with exact values nails it every time. Asked to build the whole section, it approximates.

WATCH FOR

- Referencing a shared docs file instead of pasting the spec inline — a builder should never need to go read something external.
- Bundling unrelated sections into one builder because they happen to sit near each other on the page.
- Overriding the 150-line threshold because a section “feels” simple — it's a mechanical check for a reason.

KEY TAKEAWAYS

- Every component gets a spec file with six required sections, written before any builder is dispatched.
- “N/A” is written deliberately after checking, never left blank by default — footers often still have hover states.
- The ~150-line complexity budget is mechanical: exceed it and the section gets split, no exceptions for “it's all related.”
- A builder receives its spec inline, its screenshot path, its shared imports, its target file, and a compile-check instruction — nothing else, nothing less.
- Dispatch doesn't wait — extraction moves to the next section immediately while builders work in parallel worktrees.
- Splitting a complex section isn't overcaution; it's what keeps each builder's output exact instead of approximate.

EXERCISE

Try This

Self-Reflection

Pick a section from the site you've been using across this book's exercises. Would it need one builder or several under the complexity budget?

Collaborate

Write a full six-part spec file for that one section, using real getComputedStyle() values from your browser's dev tools.

Reflection

Did writing the full spec reveal anything about the section you hadn't noticed just looking at it?

REFLECTION

- *Why is a mechanical line-count threshold more reliable here than a judgment call about "how complex this feels"?*
- *What's lost when a builder is told to reference an external doc instead of receiving everything inline?*

WHAT'S NEXT

Builders finish, worktrees merge, and the build stays green — but the clone isn't done. Chapter 7 covers the pass that actually proves it: visual QA against the live site.

CHAPTER SEVEN

Visual QA and Closing the Gap

BY THE END OF THIS CHAPTER YOU'LL BE ABLE TO

- ✓ Run a structured side-by-side comparison between a clone and its source, at two viewport widths.
- ✓ Trace a visual discrepancy back to its spec file and know which of two things actually went wrong.
- ✓ Recognize the difference between a clone that screenshots well and one that behaves correctly.

Dana's first Cowork-driven rebuild looked finished the moment the build went green. It wasn't. Northlight's real site had a testimonial carousel that auto-advanced every six seconds; the clone had a static row of three cards that happened to match the first frame perfectly. A passing build and a matching screenshot told her nothing about behavior — only the QA pass she ran afterward caught it.

Why a green build isn't the finish line

Every builder verifies npx tsc --noEmit before finishing, and every merge gets checked against npm run build — but both of those only confirm the code compiles, not that it matches the source. The skill's own instructions are explicit that visual QA is a separate, mandatory phase, not an optional polish pass: after full assembly, you do not declare a clone complete until this loop has run and come back clean.

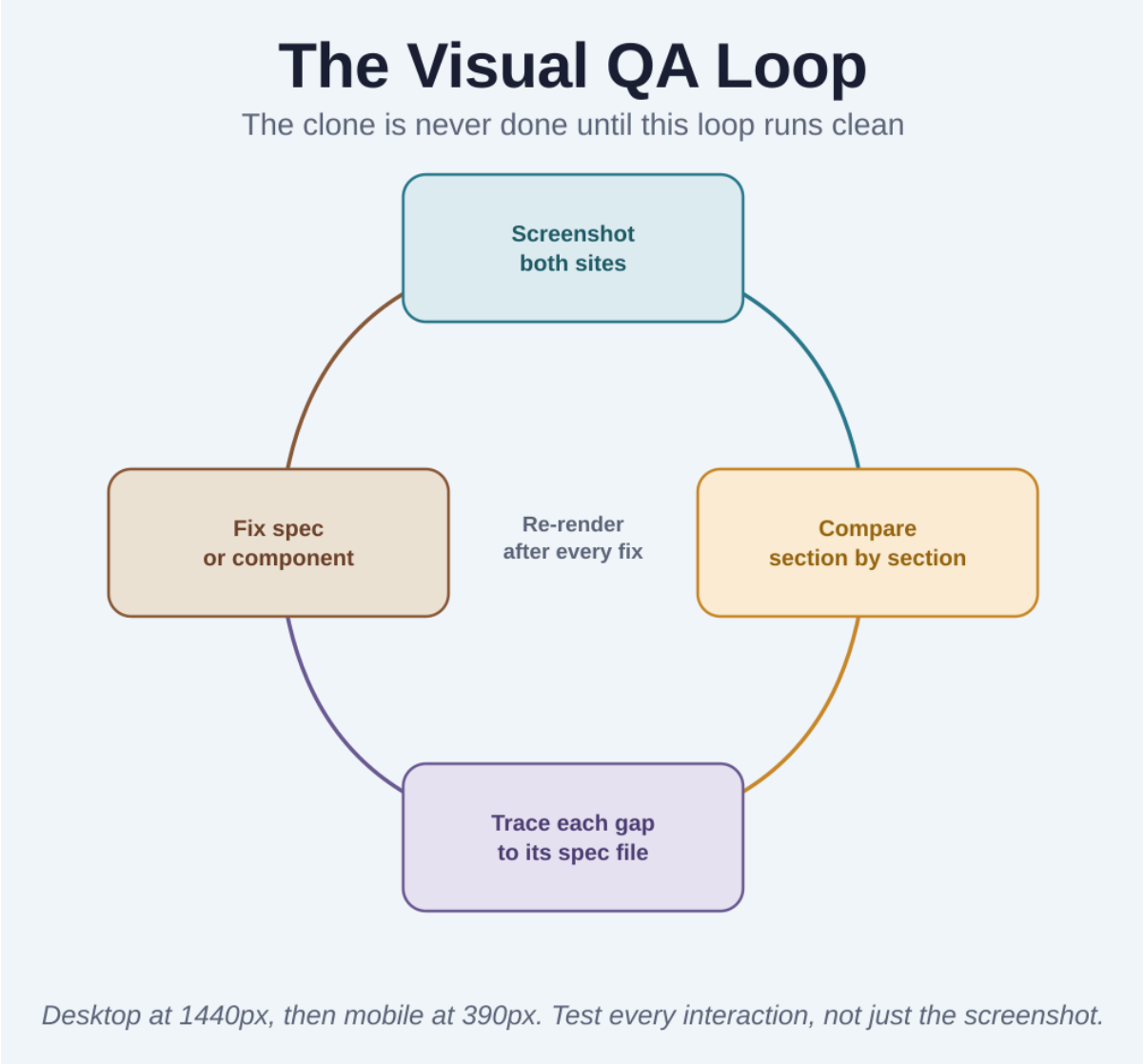

Figure 7. *Screenshot, compare, trace, fix, re-render — and repeat until the loop closes clean rather than running once.*

Running the comparison

The pass itself is structured, not a casual eyeball check. Open the original site and the running clone side by side, or capture screenshots of both at matching viewport widths. Compare section by section, top to bottom, first at 1440px, then again at 390px — a discrepancy that only shows up on mobile is exactly the kind reconnaissance's responsive sweep was supposed to catch, and re-checking here is how you find out whether it actually did.

For every gap found, there are exactly two possible causes, and telling them apart matters: either the spec file itself was wrong — a value mis-extracted back in Chapter 6 — or the spec was correct and the builder simply didn't implement it faithfully. The fix is different in each case. A wrong spec means going back to the live site, re-extracting with browser MCP, correcting the spec file, and then fixing the

component to match the corrected spec. A wrong implementation against a correct spec means fixing the component directly — the spec doesn't need to change, the code does.

Interactive behavior gets the same scrutiny as static appearance. Scroll through the finished clone and confirm smooth scroll feels right if the original had it, that header transitions trigger at the correct scroll position, that every tab switches correctly, and that any animation plays the way it did in reconnaissance's interaction sweep. A clone that matches every screenshot pixel-for-pixel but whose scroll-triggered header never actually triggers has failed this phase, even though it would pass a purely visual diff.

THE METHOD

The QA Loop

1. Screenshot the original and the clone at 1440px.
2. Compare section by section, top to bottom, noting every discrepancy.
3. Repeat the comparison at 390px.
4. For each discrepancy: check whether the spec file was correct. If not, re-extract and fix the spec first.
5. If the spec was correct, fix the component directly to match it.
6. Test every click, hover, and scroll-triggered behavior found during reconnaissance's interaction sweep.
7. Re-render and re-inspect after every fix — don't assume one fix didn't disturb its neighbor.

A clone is never done until this loop runs clean — a passing build only proves the code compiles.

THE TWO-CAUSE QUESTION

Every visual gap has exactly one of two causes: a wrong spec, or a spec that was right but poorly built. Diagnose which one before you touch any code — fixing a component that was correctly following a wrong spec just moves the error somewhere new.

WATCH FOR

- Declaring a clone finished the moment npm run build passes, without ever running a side-by-side comparison.
- Fixing a component's code when the actual problem was a wrong value in its spec file — the error resurfaces the next time that spec gets reused.
- Checking desktop and forgetting to re-run the whole comparison at mobile width.

KEY TAKEAWAYS

- A green build and a matching screenshot only prove the code compiles — neither proves behavior matches the source.
- Visual QA compares section by section, at both 1440px and 390px, against the live original.
- Every discrepancy traces to one of two causes: a wrong spec, or a correct spec that was built incorrectly — diagnose before fixing.
- Interactive behavior — scroll triggers, tab switches, hover transitions — gets tested directly, not inferred from a static screenshot.
- Fixes get re-rendered and re-inspected, since one fix can disturb a neighboring section.
- The skill's own instructions treat this phase as mandatory, not optional polish — a clone isn't complete without it.

EXERCISE

Try This

Self-Reflection

Think of a time a piece of work looked finished because it passed an automated check, but turned out not to actually work. What check was missing?

Collaborate

Take any clone or rebuild you've done — from this book's exercises or otherwise — and run the six-step QA loop against it deliberately, including the interactive-behavior check.

Reflection

Which category did your discrepancies fall into more often — wrong specs, or specs built incorrectly? What does that tell you about where your own process needs more rigor?

REFLECTION

- *Why does a passing automated build create a false sense of completion, and how does this phase correct for that?*
- *What's the risk of fixing a component's code when the real problem was upstream, in its spec?*

WHAT'S NEXT

Part III puts the whole pipeline to work on two real-shaped jobs — starting with Priya's marketing-page migration in Chapter 8.

PART THREE

Real-World Playbook

Two full jobs, worked start to finish, and the boundary the tool doesn't cross

CHAPTER EIGHT

Case Study: Cloning a SaaS Marketing Page

BY THE END OF THIS CHAPTER YOU'LL BE ABLE TO

- ✓ Follow one clone from first URL to merged, QA'd result.
- ✓ See the complexity budget and the interaction-model check applied to real, specific sections.
- ✓ Recognize how the five phases from Chapter 2 actually interleave in a working session.

Priya's bakery client finally says yes, and the target is worse than the pitch deck suggested. The marketing homepage has a sticky nav that shrinks on scroll, a hero with a looping background video, a three-tab “How It Works” section that switches on scroll rather than click, a four-card feature grid, a testimonial carousel, a pricing table, and a footer with newsletter signup. Seven sections, and by Chapter 6's rule, at least two of them won't survive as a single builder each.

Reconnaissance turns up the first surprise

Priya runs /clone-website against the staging URL and lets the interaction sweep do its work before touching anything herself. The scroll sweep catches the nav shrinking at roughly 80px of scroll — background going from transparent to solid white, a shadow appearing, padding tightening — and it catches the “How It Works” tabs switching automatically as each panel scrolls into view, driven by an IntersectionObserver rather than any click handler. That single finding saves the exact rebuild Theo had to redo twice in Chapter 5: the tab section gets documented as scroll-driven from the start, never touched with a click handler at all.

The click sweep finds something reconnaissance's screenshot alone would have missed entirely: the hero's “background video” is actually two layered elements, a muted looping video underneath and a semi-transparent gradient overlay on top, with a small play/pause icon that only appears on hover. Chapter 2's “layered assets matter” principle exists precisely for this case — a section that reads as one visual is often two or three elements stacked, and missing the overlay makes a rebuild look flat even when the base layer is pixel-perfect.

Foundation, then the split

With reconnaissance complete, foundation runs once: the client's actual fonts (a geometric sans for headings, a warmer serif for body copy) get wired into layout.tsx, their real color palette replaces the

shadcn defaults in globals.css, and a download script pulls every real image and the hero's background video into public/. Only then does section-by-section work begin.

The nav, the hero, the pricing table, and the footer each stay under the complexity budget and get one builder apiece. The feature grid — four cards, each with a distinct icon, a hover-triggered color shift, and slightly different internal spacing — crosses 150 lines the moment all four cards' exact values get written into one spec, so it becomes five specs: one per card, one for the wrapper. The testimonial carousel gets split differently, into a carousel-shell spec covering the auto-advance timing and dot indicators, and a testimonial-card spec covering the repeated internal layout, because a single card design repeats with different content rather than four genuinely different designs.

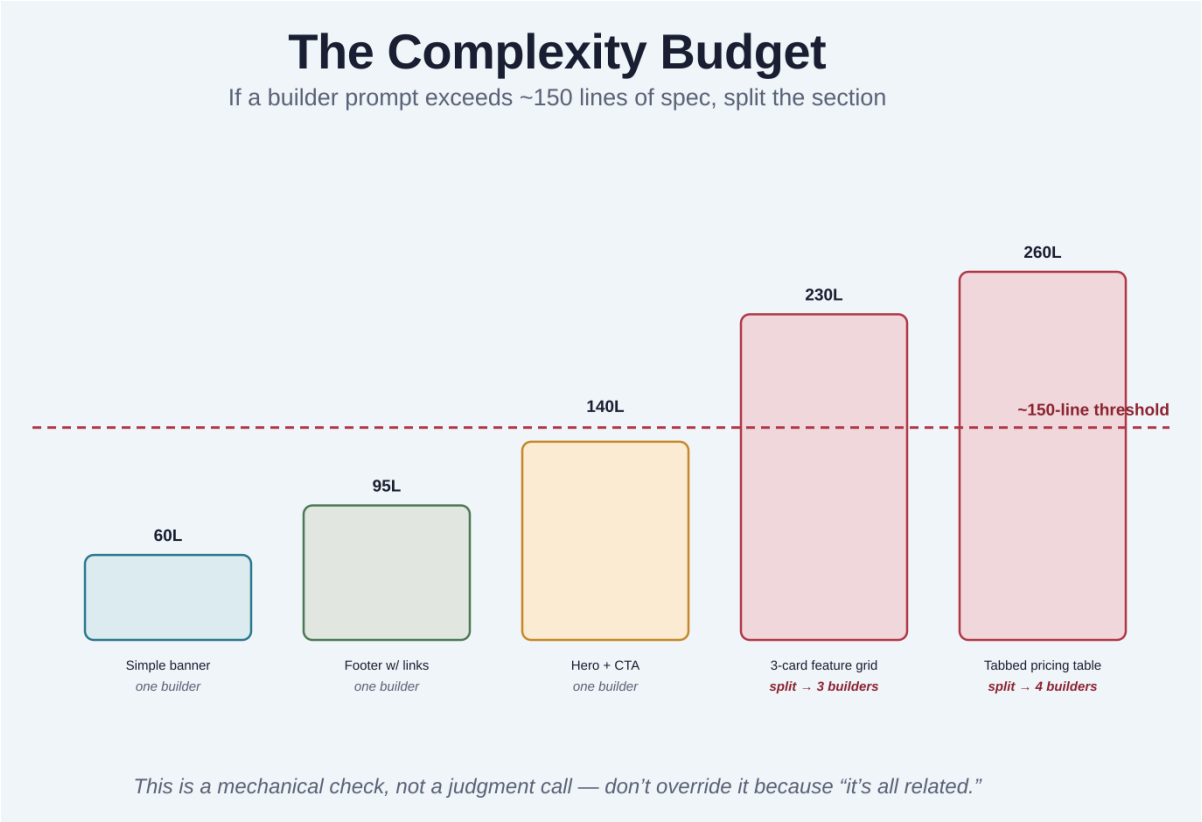

Figure 6. *Priya's actual split: four sections stay whole, two cross the line and get broken into shell-plus-repeating-card pairs.*

Merge, then the pass that catches the real gap

Builders finish across their worktrees at different times; Priya merges each as it lands and reruns npm run build after every merge, catching one type error where a card component expected a prop the wrapper wasn't passing. By the time all seven sections are merged, the build is green — and the visual QA pass still finds two real gaps: the newsletter footer's input field is missing a focus-state outline the original had, and the testimonial carousel's auto-advance timer is running at four seconds instead of the original's six, a detail that was in the spec correctly but got typed wrong in the builder's setTimeout call.

Both trace to Chapter 7's second failure category — a correct spec, an imperfect build — and both get fixed in the component, not the spec.

Seven sections, and by the complexity budget, at least two of them were never going to survive as a single builder each.

WHAT THE CLONE DOESN'T INCLUDE

Real newsletter signup still needs a backend endpoint; the pricing table's "Subscribe" buttons link nowhere yet. The visual and interaction layer is done — wiring it to Fernwood's actual services (Chapter 9's territory, not this one) is separate work the clone was never meant to replace.

WATCH FOR

- Assuming a hero "background video" is a single element before checking for a layered overlay.
- Merging every builder at the end instead of merging — and rebuilding — as each one finishes.
- Treating a passing build as done, when the actual bug (a mistyped timer value) only shows up in behavior, not in compiled code.

KEY TAKEAWAYS

- A scroll-driven tab section correctly identified in reconnaissance never gets touched with a click handler — the model is set once and holds.
- A visual that looks like one element is often layered — checking for overlays before specifying a section prevents a flat rebuild.
- The complexity budget doesn't split evenly; a repeating card design splits into a shell spec plus a single card spec, not one spec per instance.
- Merging happens continuously as builders finish, not all at once at the end, catching integration errors closer to their source.
- A green build after all merges still isn't the finish line — the QA pass caught two real, behavior-level gaps a compiler couldn't.
- A finished clone's visual and interaction layer is genuinely done; backend wiring for real functionality is separate, later work.

EXERCISE

Try This

Self-Reflection

Of the seven sections in this case study, which would you have guessed needed splitting before reading how it actually went?

Collaborate

Pick a real SaaS marketing page you know well. Sketch its section list and guess, before checking, which sections are scroll-driven versus click-driven.

Reflection

Where did your guesses in the exercise above diverge from what a real interaction sweep would likely find?

REFLECTION

- *What made the nav-shrink and tab-switch behaviors easy to miss without a dedicated scroll sweep?*
- *How does merging continuously, rather than all at once, change what kind of bugs you catch and when?*

WHAT'S NEXT

Chapter 9 follows a harder case: a multi-page legacy site with no living owner to ask questions of, and the decisions that changes.

CHAPTER NINE

Case Study: Migrating a Legacy Multi-Page Site

BY THE END OF THIS CHAPTER YOU'LL BE ABLE TO

- ✓ Scope a multi-page clone using multiple target URLs and per-site extraction folders.
- ✓ Handle a site with inconsistent patterns across pages — a common legacy-site reality.
- ✓ Decide, with evidence, when a discrepancy is a bug versus an intentional difference worth preserving.

Fernwood Analytics' marketing site was built by a contractor who left the company two years ago, and nobody has the source. Marcus Webb has the live site, an expired hosting invoice, and a board deck due in six weeks that needs the whole thing rebuilt on infrastructure the company actually controls. There's no design file to check against — the live site is the only surviving spec, which makes this a recovery job, not a redesign, and raises the stakes on getting reconnaissance right the first time.

Scoping more than one page at once

The skill accepts more than one URL in a single command, and Marcus gives it four: the homepage, a product page, a pricing page, and an about page. Multiple targets get processed independently, each with its own isolated extraction folder — docs/research/<hostname>/ per site when domains differ, or a clear per-page split when they share one domain — so a spec file for the pricing page's hero never gets confused with the homepage's hero, even though both are called “HeroSection” in casual conversation. Marcus confirms running all four in parallel rather than sequentially, since nothing about the pages depends on each other and the machine has room for it.

What a legacy site actually looks like up close

Reconnaissance surfaces the site's age immediately: the homepage and the about page share a nav component pixel-for-pixel, but the pricing page's nav is subtly different — two extra pixels of padding, a slightly different hover color — the unmistakable fingerprint of a page built in a later sprint by someone who eyeballed the original instead of reusing it exactly. This is a real decision point the skill's documentation doesn't resolve for you: is the pricing nav's drift a bug worth fixing to match the rest of the site, or a real, if accidental, feature of how the live site actually looks today that a faithful clone should preserve?

Marcus treats it as a faithful-clone question and preserves the drift — the pricing page's nav gets its own spec file with its own slightly different values, rather than being silently reconciled to match the homepage's. The scope defaults are explicit that a clone should be pixel-perfect to what's actually visible at the URL; “what the site probably meant to look like” is a design opinion the skill isn't asked for unless someone requests it.

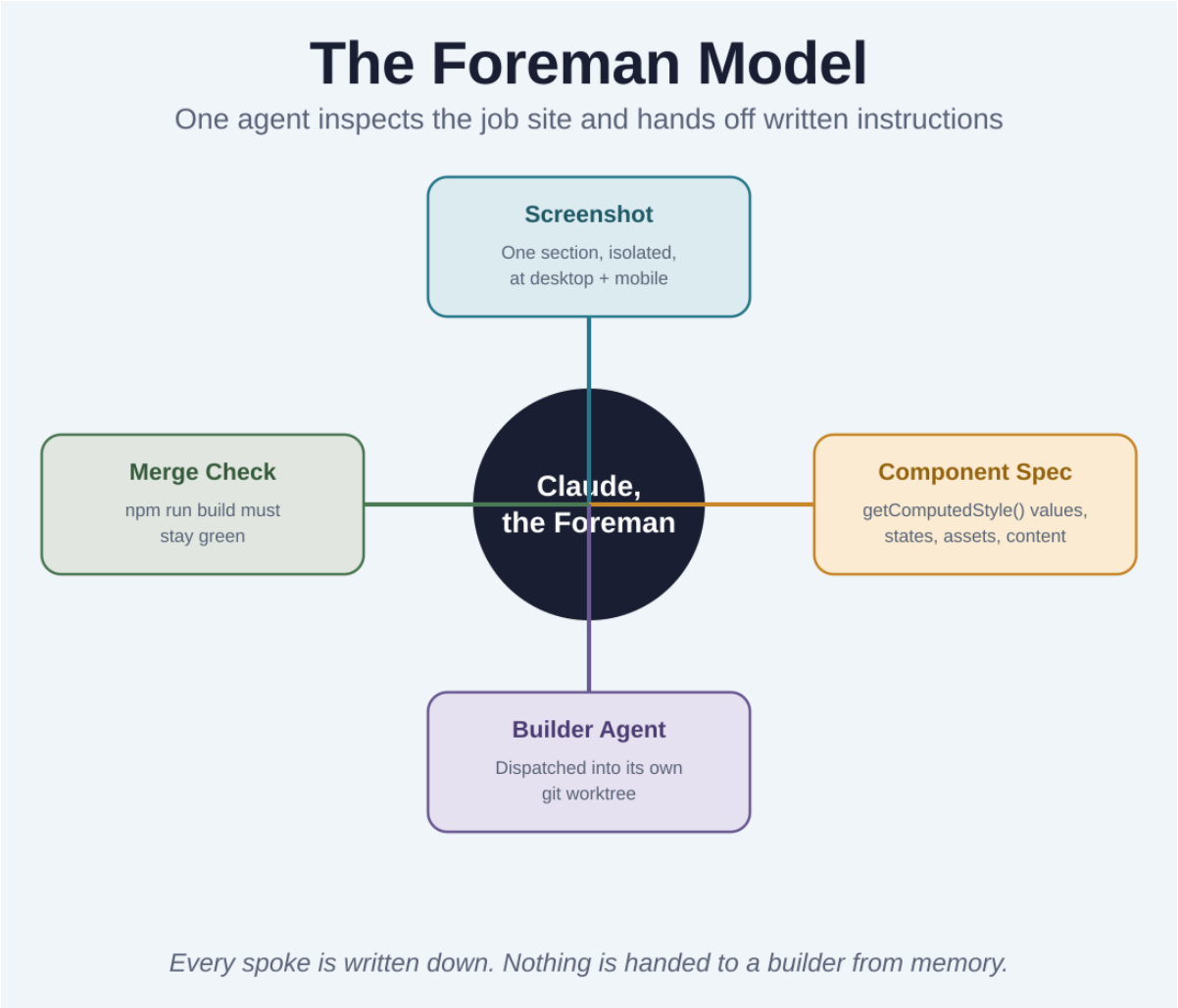

Figure 2. *Four pages, four foremen's worth of extraction — each spec file traces back to its own page, never assumed shared with a sibling.*

The parts that don't come back

Reconnaissance also turns up what the clone categorically won't include, and naming this early avoids a false promise to the board. The scope defaults rule out a real backend or database, real authentication, and real-time features by default — the product page's “live usage counter” gets rebuilt with realistic mock data, not a real data connection, because that's explicitly out of scope unless the user asks otherwise. Marcus flags this to his own team before the six-week deadline gets treated as a promise the clone alone can keep: the visual and structural migration is real work finished on schedule; wiring the counter to Fernwood's actual analytics pipeline is a second, separate project.

> THE METHOD
>
> ### Multi-Page Scoping Checklist
>
> 1. List every target page and confirm each is reachable and worth cloning independently.

2. Confirm whether pages share components exactly, or only appear to — check computed styles, don't assume from a glance.
3. Decide explicitly: preserve inconsistencies as found, or flag them for a follow-up design pass. Document the decision.
4. Name what's structurally out of scope — real backend, auth, live data — to stakeholders before the deadline conversation, not after.
5. Run visual QA per page, not just once across the whole site — a shared component with per-page drift needs per-page verification.

The live site is the only surviving spec, which raises the stakes on getting reconnaissance right the first time.

A GENUINELY HARD CALL

There's no universally correct answer to nav-drift-preserve-or-fix. What matters is that the decision gets made deliberately, with the scope defaults in view, and documented — not silently absorbed into whichever way a builder happened to interpret an ambiguous spec.

WATCH FOR

- Assuming shared-looking components across pages are actually identical without checking computed styles page by page.
- Letting a mock “live counter” or similar placeholder quietly get presented to stakeholders as functioning data.
- Running visual QA once across a multi-page site instead of once per page — per-page drift needs per-page checking.

KEY TAKEAWAYS

- Multiple URLs can be passed to the skill in one command; each gets its own isolated extraction folder to avoid cross-contamination.
- A recovery job — no source, only a live site — makes reconnaissance's accuracy the entire foundation of the project.
- Legacy sites often have components that look shared but aren't identical; check computed styles per page rather than assuming.
- Preserving an inconsistency versus fixing it toward consistency is a real decision the skill won't make for you — make it deliberately.
- Real backend, auth, and live-data features are out of scope by default; mock data fills their place unless told otherwise.

- Naming what's out of scope to stakeholders early prevents a finished visual migration from being mistaken for a finished product.

EXERCISE

Try This

Self-Reflection

Have you inherited a project where two parts that looked identical turned out to have quietly drifted apart? How did you find out?

Collaborate

If you have access to a multi-page site, compare the same component — a nav, a footer — across two different pages using dev tools. Look for small, easy-to-miss differences.

Reflection

If you found drift, would you preserve it or fix it in a rebuild — and what would you need to know to decide?

REFLECTION

- *What made this project's stakes different from Priya's in Chapter 8, and how did that change Marcus's decisions?*
- *Why is naming out-of-scope items early a form of risk management, not just a technical footnote?*

WHAT'S NEXT

The final chapter turns from technique to judgment — what goes wrong most often, and the line the skill's own documentation draws around what it should never be used for.

CHAPTER TEN

Troubleshooting, Ethics, and What Not to Do

Every practitioner in this book has, by now, broken a clone in a way the previous nine chapters could have prevented. That's not a flaw in the tool — it's what a genuinely mechanical, exhaustive process looks like from the inside, and the fixes are, without exception, things the skill's own documentation names directly as lessons learned from failed clones that cost real hours of rework.

The troubleshooting list, and why each item costs what it costs

Building a click-based interface when the original is scroll-driven, or the reverse, is the single most expensive mistake in the entire pipeline — not a CSS fix but a full rewrite of a section's interaction logic, which is exactly why Chapter 5 insists on scrolling before clicking, every time, no exceptions. Extracting only a component's default state is the second most common failure: a tab bar showing “Featured” on page load still has three other tabs' worth of content waiting behind clicks nobody made during reconnaissance, and a header that only gets captured at scroll position zero never reveals what it looks like at position two hundred.

Missing layered or overlay images is a quieter failure with the same root cause as the hero video in Chapter 8: a container's DOM tree can hold two, three, or more images stacked — a background gradient, a foreground mockup, an absolutely-positioned icon — and a clone that only captures the bottom layer looks emptier than the original in a way that's hard to name but easy to notice. Building an elaborate HTML mockup of something that's actually a video, or a Lottie animation, or canvas content wastes real effort reconstructing motion that a `<video>` tag would have handled directly — always check the element type before building around what something merely appears to be.

Approximating a CSS value — calling something text-lg because it “looks like” 18px, when the actual computed line-height doesn't match that utility class's default — is Chapter 2's first principle violated in miniature, and it compounds: one approximated value in a spec becomes one silently wrong value in every builder that ever reads that spec again. Dispatching a builder without a spec file at all, skipping asset extraction because “the CSS is what matters,” or referencing an external doc from inside a builder's prompt instead of pasting values inline all trace back to the same shortcut — treating extraction as overhead instead of as the actual work.

> **THE FULL LIST, AT A GLANCE**
>
> - Click before scroll — always test scroll-driven behavior first.
> - Default-state-only extraction — click every tab, capture every scroll position.
> - Missed layered images — check every container for stacked or overlay elements.
> - Mockups built for actual video/canvas/Lottie content — check the element type first.

- Approximated CSS classes — use the real computed value, not a guessed utility class.
- Builders dispatched without a spec file, or with references to external docs instead of inline values.
- Skipped responsive testing — test at 1440px, 768px, and 390px, not desktop alone.
- Forgotten smooth-scroll libraries — default scrolling feels noticeably different and users notice immediately.

Where the boundary actually sits

The skill's own README is direct about scope, and this book has followed that framing throughout: legitimate uses are platform migration, recovering a site whose source code is lost, and studying how production sites achieve specific effects. It is explicitly not intended for phishing or impersonation, for passing off someone else's design and brand assets as your own, or for violating a site's terms of service where those terms prohibit scraping or reproduction. A pixel-perfect clone of your own site and a pixel-perfect clone built to impersonate someone else's can look identical in a code review — the difference is entirely in who owns what's being rebuilt and what it's used for afterward, and that difference doesn't show up anywhere in the code.

This isn't a footnote to bolt on at the end; it's the frame Chapter 1 opened with. Clone your own sites freely. Clone a site you have explicit permission to migrate. Clone a competitor's landing page to study a pattern, the way Theo does throughout this book, and keep that study private and educational rather than shipping the result as your own product. When in doubt about whether a specific use crosses the line, the honest test is simple: would the site's actual owner be comfortable knowing this clone exists and what it's being used for? If the answer is no, or even uncertain, that's the answer.

A migration clone and an impersonation clone can look identical in a code review. The difference is ownership and intent — and it never shows up in the code.

A PRACTICAL HABIT

Before running the skill against any site you don't own, spend thirty seconds checking its terms of service for language about scraping or reproduction. It's a small habit that catches the cases this chapter can't enumerate in advance.

KEY TAKEAWAYS

- Every item on the troubleshooting list traces back to one of Chapter 2's nine principles, violated somewhere in the pipeline.

- Scroll-before-click ordering and full-state extraction prevent the two costliest classes of rework in the entire process.
- Layered images and video/canvas content need to be checked for directly, not inferred from how a section merely appears.
- An approximated CSS value doesn't just affect one component — it silently propagates to every spec that reuses it.
- The skill is explicitly scoped to migration, recovery, and study — never impersonation, phishing, or misrepresenting someone else's work as your own.
- A clone's legitimacy lives in ownership and intent, not in the code itself, which is exactly why it's worth deciding deliberately before you start.

EXERCISE

Try This

Self-Reflection

Look back across every exercise in this book. Which mistake from this chapter's list did you personally come closest to making?

Collaborate

Before your next real clone, write down — in one sentence — who owns the site and why you have standing to rebuild it. Keep that sentence with the project.

Stretch Goal

Audit a clone you've already built (from this book's exercises or otherwise) against the full troubleshooting list. How many items hold up?

REFLECTION

- *Which single principle from Chapter 2, if you'd internalized it first, would have prevented the most rework across this book's exercises?*
- *How will you decide, going forward, whether a given clone is inside the boundary this chapter draws?*

WHAT'S NEXT

The book closes with a one-page reference to the whole pipeline, a glossary of the terms this book has used throughout, and where to go for the skill's own source material.

BACK MATTER

The Pipeline at a Glance

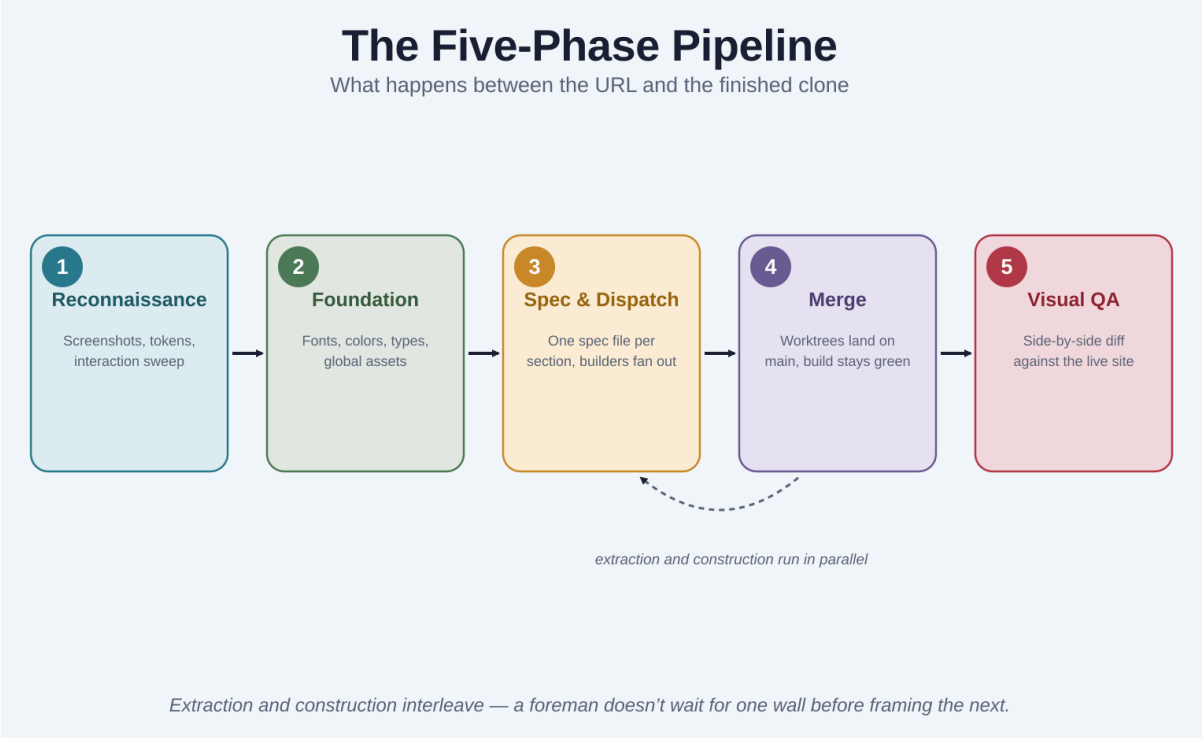

Figure 8. *The five phases, start to finish — keep this page open while you run your own first clone.*

Reconnaissance: full-page screenshots at 1440px and 390px, global fonts and colors, then the three-part interaction sweep — scroll first, then click, then hover — written to BEHAVIORS.md and PAGE_TOPOLOGY.md.

Foundation: fonts, colors, TypeScript types, extracted icons, and a real asset-download script. Sequential, done once, blocks everything after it.

Spec & Dispatch: one six-part spec file per section — Overview, DOM Structure, Computed Styles, States & Behaviors, Assets & Content, Responsive Behavior — then a builder dispatched into its own git worktree with that spec inline. Split any section over ~150 lines of spec content.

Merge: each finished worktree folds into main; npm run build re-verified after every merge, not just at the end.

Visual QA: side-by-side comparison against the live site at both viewport widths, every discrepancy traced to either a wrong spec or a wrong build, every interactive behavior tested directly — not assumed from a screenshot.

Glossary

Builder agent. An agent dispatched to implement one component or section from its written spec file, working inside its own git worktree so its changes don't collide with any other builder's.

Complexity budget. The mechanical rule that a builder prompt exceeding roughly 150 lines of spec content signals a section that needs to be split into smaller, more focused specs.

Component spec file. The six-part document — Overview, DOM Structure, Computed Styles, States & Behaviors, Assets & Content, Responsive Behavior — written for every section before any builder is dispatched.

Foreman model. The skill's own description of its process: one agent inspects the page section by section, writes down exactly what it finds, and hands that writing to specialists, rather than building everything from memory after a single inspection pass.

Interaction model. Whether a section responds to scroll, click, hover, time, or some combination — determined during reconnaissance, before any spec is written, by scrolling through a section before ever clicking it.

Interaction sweep. The dedicated reconnaissance pass that tests scroll behavior, then click behavior, then hover behavior, in that order, to catch behaviors invisible in a static screenshot.

Layered assets. Multiple images or elements stacked within one container — a background, a foreground mockup, an overlay icon — that together read as a single visual but must be extracted and rebuilt individually.

Visual QA. The mandatory closing phase comparing the finished clone against the live original, section by section, at multiple viewport widths, including every interactive behavior — not just a static screenshot diff.

Worktree. An isolated git working copy that lets multiple builder agents make changes in parallel without one agent's in-progress work interfering with another's, merged back into the main branch once each builder finishes.

Further Resources & Attribution

The AI Website Cloner skill discussed throughout this book is an open-source template published by the developer known as JCodesMore, available at github.com/JCodesMore/ai-website-cloner-template under the MIT license. The framework, the five-phase pipeline, the nine guiding principles, and the component-spec template described in this book all originate from that project's own SKILL.md and README — this book is an independent field guide to using it well, not an official product of that project or of Anthropic.

For the underlying platforms this book references: Claude Code's own documentation covers custom skills, browser automation setup, and the full command reference. Claude Cowork's documentation covers the skills directory, custom skill uploads, and the Capabilities settings that gate code execution. Both evolve; check each platform's current documentation before assuming a specific menu path or setting name still matches exactly what's described here.

None of the people or companies named in this book's case studies — Priya Shah, Marcus Webb, Dana Okafor, Theo Lindqvist, Fernwood Analytics, Northlight Studio — are real. They exist to make an abstract pipeline concrete, and any resemblance to a real person, company, or project is coincidental.