SUEDE CREATOR SKILLS · 73 OPEN-SOURCE AGENT SKILLS

S-Tier

The Builder's Book Behind the Suede Skills

How agent skills actually work, and how a builder gets good enough that the work keeps running without them.

Jason Colapietro · Suede Labs AI

29,302 words · MIT

Colophon

Title	S-Tier: The Builder's Book Behind the Suede Skills
Author	Jason Colapietro, founder of Suede Labs AI
Edition	First, August 2026
Length	29,302 words across 14 chapters, front matter, and 2 appendices
Licence	MIT. Copy it, quote it, print it, hand it to someone.
Read online	skills.suedeai.ai/book/
Source	github.com/JasonColapietro/suede-creator-skills → book/

Every claim in this book points at a file in a public repository. Checking claims is most of what the book argues for, so the citations are paths rather than footnotes: open the repo and read the skill.

The prose was written against the same anti-slop rules the pack enforces, which is why it contains no em dashes. Quoted repo material keeps its own punctuation.

Thirty-eight of the marketing and growth skills referenced here are adapted from **marketingskills** by Corey Haines under the MIT Licence. That project is the origin of the material and the credit belongs there.

Contents

FRONT MATTER

Front matter

Why the book exists, who it is for, how to read it, and the four ideas every chapter leans on.

The Builder's Book Behind the Suede Skills

By Jason Colapietro, Suede Labs AI

Companion to suede-creator-skills, a 73-skill MIT-licensed pack for Claude Code and OpenAI Codex.

Why this book exists

The pack came out of a specific irritation. A solo founder kept hiring marketing firms for his own products and kept watching them skip fundamentals that take twenty minutes: metadata that never got written, a signup flow nobody measured, a launch page with no proof on it. The fix was not a better vendor. The fix was writing the procedure down once, in a form an agent could run, and never re-explaining it.

That is what a skill is. A folder, a markdown file, a procedure with a defined output. Seventy-three public skills, readable before you install them and editable after.

This book is the reasoning underneath. Half of it is how the machinery works: progressive disclosure, description routing, agent lanes, grade cards, evidence gates. The other half is the part the machinery cannot supply, which is judgment about what to build, when to stop, and which of your habits are quietly keeping you at the tier you are on.

Who it is for

You already use an agent daily. You write decent prompts. You have noticed that output volume went up and the ceiling did not move much. That gap is the subject.

You do not need the pack installed to read this. Every chapter names real files in a public repo, so you can check the claims. Checking claims is most of what the book argues for, so please do.

How to read it

Straight through the first time. Parts I and II establish the vocabulary the rest of the book leans on, and Chapter 11 will not land without them.

- **Part I. The Machinery** (Chapters 1 to 3): what a skill is, how it is built, how an agent decides to load it.

- **Part II. The Operating System** (Chapters 4 to 6): outcome-bound work, agent lanes and worker fleets, and the verification discipline the whole thing rests on.
- **Part III. The Craft Lanes** (Chapters 7 to 10): grading code, treating design and copy as systems, distribution, and the last mile into a store or a real account.
- **Part IV. Becoming S-Tier** (Chapters 11 to 14): the ladder, the judgment layer, writing your own skills, and a ninety-day plan.

Then the appendices: a skill index organized by what you are trying to do, and the operating rules the book keeps returning to, collected in one page.

Four ideas the book repeats

Progressive disclosure. An agent reads every skill's description and almost none of their bodies. The description is the router. The body is the payload, loaded only on a match. This is why a pack can grow to seventy-three skills without drowning the context window, and why a badly written description is a broken feature rather than a cosmetic one.

Evidence or it did not ship. A claim without command output, an HTTP status, a diff, or a rendered screenshot is a guess wearing a confident tone. Agents are extremely good at the confident tone.

Bring a decision, not a workshop. Compute is authorized and abundant. Your attention is neither. Work that hands you five options and asks you to pick has usually just moved the job back to you.

The ladder. C-tier builders produce output. B-tier produce working output. A-tier produce verified output. S-tier build systems that keep producing verified output when they are not in the room.

A note on the prose

This repo ships `suede-deslop`, a skill that strips AI writing patterns from text before it goes public: filler openers, manufactured enthusiasm, false agency, formulaic contrast, metronomic rhythm, and em dashes. It would be strange to ship that skill and then publish a book that trips every rule in it.

So the book was written under those constraints. The prose carries no em dashes. Quoted material from the repo keeps its own punctuation, because changing a quotation to fit a style rule is the kind of small dishonesty this pack exists to catch. If you find a violation anywhere else, the file to check is `skills/suede-deslop/SKILL.md`.

License and credit

PART I

The Machinery

What a skill is, how it is built, and how an agent decides to load it.

PART I · CHAPTER 1

The Competence Gap

A capable model plus a vague prompt produces confident mediocrity. The gap is not intelligence, it is procedure, and procedure is now a file you own.

You paste a diff into Claude Code and type "review my diff." Thirty seconds later you get a tidy response. Two naming suggestions, one note about extracting a helper, a closing line that the change looks solid. You merge it.

The diff added a `role` check to an API route. The agent never opened the middleware that route sits behind. It never checked whether the role came from the session or from a request parameter. It never ran your test suite, and it never said it hadn't. It read the changed lines, pattern-matched them against a general sense of what good code looks like, and reported back with the tone of someone who checked.

That is the competence gap. The model was smart enough to catch the bug. Nothing in the request told it to go look.

Confident mediocrity has a shape

When a capable model gets a vague instruction, it does not fail randomly. It fails in a consistent, recognizable way, and once you see the shape you cannot unsee it.

First, it picks a plausible approach instead of the correct one. "Review my diff" has no defined scope, so the agent invents one, and the scope it invents is almost always the changed lines. Callers, config, migrations, and the `.env.example` that no longer matches the code are outside the frame it drew for itself.

Second, it skips the checks a specialist would run without being asked. A senior reviewer looking at an auth change does not check the happy path. They check the expired token, the missing token, the token signed with `alg: none`, and the role field an attacker can put in a query string. That list is not intelligence. It is procedure, learned by watching those exact things go wrong.

Third, and worst, it reports success it never verified. "Tests should pass" and "tests pass" are one word apart in English and infinitely far apart in fact. A model with no instruction about evidence will produce the confident version, because confident prose is what most of its training data looks like.

None of these are reasoning failures. Ask the same model directly whether a role check read from `req.query` is safe and it will tell you no, at length, with a correct explanation. The knowledge is there. The procedure that would have made it look is not.

Why a better prompt does not scale

The obvious fix is to write a better prompt. So you do. You write four paragraphs specifying that the review should trace callers, check the auth boundary, run the repo's linter and test suite, and report which checks it could not run. It works. The review is genuinely good.

Then you close the terminal.

Tomorrow the diff touches a Stripe webhook, and the four paragraphs you wrote are about auth. Next week you need a design review, then a README pass, then a migration check. Each one needs its own four paragraphs, written under time pressure, by you, from memory, at the exact moment you least want to be writing process documentation.

There is a second failure that hurts more. The prompt you wrote yesterday was good because you remembered eleven things. Today you remember nine. The two you dropped are invisible, because a missing check produces no output. You cannot review a prompt for the checks that are not in it. Prompting well is a skill that degrades silently under fatigue, and it does not survive being handed to anyone else, including yourself in a month.

The scaling problem is not that prompts are hard to write. It is that a prompt is a thing you say, and things you say are not versioned, not reviewable, and not reusable across a hundred tasks.

What a skill actually is

Mechanically, a skill is boring, which is the best thing about it.

It is a folder with a `SKILL.md` file inside. In this repo they live at `skills/<name>/SKILL.md`, 73 of them, MIT licensed. The file opens with YAML frontmatter carrying a `name` and a `description`. Everything after the closing `---` is the body: the procedure the agent reads when the skill fires.

```
---
name: suede-deslop
description: "Suede Labs anti-slop pass: strip AI writing patterns from prose before anything
goes public. ..."
---
```

That is the whole mechanism. No plugin API, no compilation step, no runtime. A skill is a document, and it works because the agent reading it is a competent reader who follows written instructions.

The body is where the specialist lives. `skills/suede-code-grader/SKILL.md` is 212 lines. It contains a list of instant-F triggers checked before any lane is scored, seven grading lanes, grade caps that hold auth and payment changes at C without named bypass-path evidence, and a fixed output block ending in a ship gate. `skills/suede-deslop/SKILL.md` carries a 25-row kill list of filler phrases with

replacements, ten formulaic structures with their fixes, and a 50-point score with a threshold below which the text does not ship. None of that is clever. All of it is written down, which means it runs the same way on a Tuesday afternoon as it does at midnight.

Progressive disclosure, or why 73 skills fit

Here is the objection that arrives immediately. The 73 `SKILL.md` files in this repo total 1,054,432 bytes. Loading all of them into every conversation would crowd out the thing you actually came to do.

They are not all loaded. Only the frontmatter descriptions stay resident, and all 73 descriptions together come to 43,912 bytes. That is roughly a twenty-fourth of the corpus. The agent holds a catalog of what exists and reads a body only when a description matches the task in front of it.

This is progressive disclosure, and it changes what a description is for. The description is not marketing copy for the skill. It is the router. It carries the entire burden of deciding whether 400 lines of procedure load or stay on disk. Chapter 2 takes that apart in detail, including the `NOT FOR:` convention that 71 of the 73 descriptions in this repo use to push work toward a better match.

The consequence for you is direct: a skill you install and never trigger costs you almost nothing, and a skill with a sloppy description is invisible no matter how good its body is.

The before and after

Take the same diff and invoke `suede-code` instead of asking for a review.

The skill's body defines the target before any judgment happens: repo, branch, commit range, what the change claims to accomplish, and which risk lanes it touches. Then it builds a context graph, five specified layers, including "imports, callers, routes, API handlers, jobs, hooks, models, schemas, and config/env dependencies touched by the change." The scope stops being something the agent improvises.

Before manual analysis, it runs the gates your repo already ships: typecheck, the configured linter on changed files, the test suite, the dependency auditor when dependencies changed, a real secret scanner over the diff. The instruction is explicit about the failure mode it exists to prevent:

```
Detect what exists; run only that; never fabricate a result you did not run,
and note in Verification when a gate could not run.
```

Then it reads your `CLAUDE.md` and `AGENTS.md` and treats them as binding, so it stops flagging the patterns your team already decided to accept.

Only then does it review. Step 1 checks instant-F triggers, a fixed list where any single match locks the grade at F: hardcoded secrets, SQL built by string concatenation, auth middleware with a path that skips it, a webhook with no signature verification, a migration with a DROP and no tested restore. Step

2 runs language traps by stack, so a React diff gets checked for stale closures in a `useEffect` with an empty dependency array and a Swift diff gets checked for escaping closures capturing `self` strongly. Step 3 runs the OWASP Top 10, but only when the diff imports crypto, session, or payment modules, or touches auth, api, middleware, or routes.

The output has a defined shape, ending in two lines that are not opinions:

```
Overall: A–F
Grade cap applied: [surface type] — [what evidence would lift the cap] | none
...
Ship gate: ship | ship-with-caveats | hold
```

The ship gate follows the grade mechanically. A becomes `ship`, B becomes `ship-with-caveats`, and C, D, and F all become `hold`. There is no room for the agent to be encouraging.

The skill even anticipates the ways a review talks itself into a better verdict, and names them in its Red Flags section. The first is "CI passed, round up." The second is "The work was clearly hard." Effort never moves a grade. Evidence does.

The difference between the two runs is not that the second agent is smarter. It is the same model. The difference is that the second one was handed a procedure that had already made the mistakes, and had the corrections written into it.

The thesis

Your agent will do roughly what a competent professional would do given the same instructions. That is the whole game. The instructions are the variable, and for most people the instructions are whatever they happened to type.

The ceiling on your output is now the quality of the procedures you can hand an agent. That is a real ceiling, and it is not set by the model. It is set by how much hard-won practice you have managed to write down in a form something else can execute.

The good news is that those procedures are files. You can read them, diff them, argue with them, delete the parts that are wrong, and commit the parts that survived a real incident. A model you cannot inspect got better at reasoning. A folder you own got better at remembering. The second one is the part you control.

> **THE MOVE**
>
> Take the last task you prompted well and got a good result from, and write the prompt down as a `SKILL.md` body before you forget which eleven things you asked for.

PART I · CHAPTER 2

Anatomy of a Skill

A real SKILL.md taken apart: frontmatter, routing description, body, bundled references and scripts, and the advisory-gate policy that keeps a grader from blocking you.

Open `skills/suede-code-grader/SKILL.md` and read it top to bottom. It takes about six minutes. At the end of those six minutes you will know exactly what the agent will do to your diff, which parts it will refuse to do, and what the output will look like before you run it once.

That property is the point of the format. A skill is a document you can audit before you trust it. This chapter takes one apart.

The frontmatter is two fields

Every `SKILL.md` in this repo opens the same way:

```
---
name: suede-code-grader
description: "Suede Labs AI blunt A–F ship grade for a code change across correctness, security
and permissions, data and state, domain truth, UX and release behavior, tests and verification,
and deploy readiness, with Instant-F triggers and evidence-based grade caps on auth, payment,
migration, and public-API surfaces. Use when asked to grade this, give it a letter, is this an
A, how ready is this to ship, or should this merge — when the caller wants the verdict without
a findings list. NOT FOR: findings, evidence, and fix briefs (use suede-code-review, or suede-
code for findings plus grade); enforcing the verdict in CI (use suede-ci-gate); eval coverage
for AI behavior (use suede-ai-eval)."
---
```

Across all 73 skills, the frontmatter carries `name` 73 times and `description` 73 times. Forty-five of them add a `metadata` block with a version string, a convention inherited from the marketing skills adapted from Corey Haines's `marketingskills` under MIT. Nothing else appears. The schema is small on purpose, because everything the frontmatter does happens before the body is read.

`name` matches the folder and is the handle. It is what you type when you invoke the skill directly, and what other skills reference when they route work elsewhere. Keep it stable. A rename breaks every routing line in every other skill that points at it, and those lines are plain text, so nothing will tell you.

The description carries the routing burden

The description is the only part of a skill that stays in the agent's context when the skill is not running. In this repo the 73 files total 1,054,432 bytes; the 73 descriptions together are 43,912. The agent holds the small number and reaches for the large one only when a description matches.

That makes the description a router, not a summary. It has to answer two questions well enough that an agent mid-task can decide in one pass: what does this produce, and when should it fire.

Look at what `suede-code-grader` does with 686 characters. "Suede Labs AI blunt A-F ship grade for a code change" states the artifact. The lane list names the surfaces it covers. The trigger list catches the request in the words a caller actually types: "give it a letter", "is this an A", "should this merge". Then the last sentence does the real work: "NOT FOR: findings, evidence, and fix briefs (use suede-code-review, or suede-code for findings plus grade)." That sentence exists because `suede-code-review` lives next door and produces something different, and without it an agent would pick between them by coin flip.

`suede-deslop` does the same job in 512 characters:

```
description: "Suede Labs anti-slop pass: strip AI writing patterns from prose
before anything goes public. Em dashes, filler openers, manufactured enthusiasm,
false agency, passive voice, formulaic structures, all of it. Use when copy, a
README, an email, a social post, or a doc is about to ship, after a long
AI-assisted writing session, or when text sounds fine but feels generated. NOT
FOR: writing new copy (use suede-copy); changing or certifying facts, which must
be checked against primary evidence before publication."
```

Three parts. What it does, the trigger conditions in the vocabulary a user actually types ("sounds fine but feels generated"), and then the boundary.

The `NOT FOR:` convention

Seventy-one of the 73 descriptions in this repo end with a `NOT FOR:` clause, and it is the highest-leverage sentence in the file. The remaining two close the same way without the colon, which is drift rather than a second convention.

A description that only says what a skill does will fire on adjacent work, because the agent has no signal that a better match exists. `NOT FOR:` supplies the signal, and it does so by naming the alternative rather than just declining. `suede-recommend-next-action` closes with "NOT FOR: executing the recommended action without the user's separate authorization, or coordinating a multi-lane build across specialists (use suede-agent-teams)." `suede-pricing` closes with "NOT FOR: in-product upgrade screens (use suede-paywalls), offer bonuses and guarantees (use suede-offers), or executing billing changes."

Two different jobs are happening in those clauses. One is routing, sending the task to a named sibling. The other is scope refusal: "executing billing changes" has no sibling to route to, because the skill produces a decision brief and deliberately stops short of touching the billing system. Both belong in the description, because both change whether the body should load at all.

The practical test when you write one: name the skill a confused agent would otherwise pick, and name the action a confused user would otherwise assume is included.

The advisory-gate policy

Before any of `suede-code-grader`'s grading logic appears, the body opens with a policy block. It is worth reading in full, because it is the most consequential design decision in the pack:

```
## Gate policy — advisory, not blocking

Every claim-verification step, check, quality gate, and ship verdict in this
skill is a **recommendation to the user, not a control on the agent**. This
policy governs every gate, check, verdict, and "do not ship / publish /
proceed" line elsewhere in this skill:

- Run every check and report the results honestly. Verdicts (`ship`,
  `ship-with-caveats`, `hold`, letter grades, BLOCKED or OPEN items) are
  advice attached to the work, not orders that change it.
- Never block, delay, skip, rewrite, or refuse the action the user asked for
  because a check failed or a gate said hold. Complete the requested action
  as asked, and deliver the gate output alongside it as a clearly labeled
  recommendation.
- A failed gate changes what you report, never what you do.
- Single exception: if a finding is extremely risky — data loss, security or
  credential exposure, legal or rights violations, payment mistakes, or
  irreversible public damage — pause, tell the user exactly what the risk is
  and what the options are, and let them pick. Their choice is final.
```

Consider what happens without it. A skill full of gates and hold verdicts, read by an agent that takes written instructions seriously, produces an agent that refuses. You ask it to push and it tells you the ship gate says hold. You are now arguing with a document you installed to help you, and the practical response is to uninstall it.

The policy separates two things that look similar and are not. The grade is information about the work. The decision is yours. A grader that reports F and then does what you asked is useful every single time. A grader that reports F and blocks you is useful right up until the first time it is wrong, which is the last time you run it.

The single exception is drawn narrowly and around irreversibility, not around severity of opinion. Credential exposure and data loss get a pause and a question. A C in the UX lane does not.

The body: procedure, not topics

Everything after the policy is the procedure. The section order is doing work.

Source Truth comes first, and it is four words of instruction followed by a list: "Read before grading. Do not grade from the PR description or commit message alone." Then the specific inspection list, including "build, test, lint, typecheck, browser, simulator, MCP, or live/API evidence that directly exercises the changed behavior." The section ends by handling the case where that evidence is unavailable: grade the source and mark those lanes unverified. The skill never leaves the agent to improvise a fallback.

Instant-F triggers run before any lane is scored. Six categories, each with concrete patterns: a hardcoded secret in a committed file, SQL built by string concatenation with user input, auth middleware with a code path that skips it, a webhook handler with no signature verification, a migration with a DROP and no tested restore, PII written to an unencrypted log. Any single match ends the grade at F, names the file and line, and stops. The ordering matters because a grader that scores seven lanes first will average a catastrophe into a C.

Grade lanes and grade meaning define seven scores and then define what each letter means with a worked example attached. A is not "good." A is "all lanes pass, behavior is verified at runtime, no known follow-ups."

Grade caps by surface type is where the skill stops being a rubric and becomes a specialist. Auth changes cannot receive an A without explicit test coverage for the bypass path, with named evidence such as "tested with expired token returns 401." If that is missing and the happy-path caveats are missing too, the instruction is unambiguous: "If neither condition is met: cap at C regardless of other lane performance." Data migrations with no rollback plan cap at D.

Technical debt indicators carries a table that grades the same pattern differently by location, which is the kind of judgment most rubrics flatten:

PATTERN	LOCATION	GRADE IMPACT
God object (5+ unrelated concerns)	Payment module	D in Correctness
God object	Utility helper	B in Correctness

The **Red Flags** section lists the rationalizations that inflate grades, in the agent's own voice: "CI passed, round up." "The work was clearly hard." "It's just a refactor."

Output Format fixes the artifact: a plain-language summary, the target, the seven lane grades, the overall, the cap applied, the evidence, three required upgrades, and a verification block split into Checked and Not checked. The last line resolves mechanically. A becomes `ship`, B becomes `ship-with-caveats`, C and below become `hold`.

Boundaries closes the loopholes, including one rule that prevents the most annoying possible failure: "Never report a C, D, or F without naming the required upgrade that would move the grade."

Then a worked example, which is unusual and worth copying. The skill grades a real file from its own repo, `mcp/suede-skills-mcp.mjs`, 715 lines, spawned as a process and driven with real JSON-RPC requests over stdin. It lands on an overall A with a B in Suede truth over an unbounded string echo, and it closes by naming its own standard: every claim ties to a line number or a command actually run, "not an invented test result or a guessed line number."

What lives beside the SKILL.md

A skill folder can carry more than one file, and three subfolder conventions appear in this repo. `suede-agent-teams` uses all three.

`references/` holds material too long to keep in the body. `suede-deslop` keeps its 25-row table in `SKILL.md` and pushes the full sweep, forty-plus more phrases, to `references/kill-list.md`, loaded only when the text is going to press. `suede-agent-teams` keeps its 327-line public-contribution program the same way. This is progressive disclosure applied one level down.

`scripts/` holds executable work. `suede-agent-teams/scripts/contribution-ledger.mjs` is 1,286 lines of real code, because atomic issue leases are a thing a program should do deterministically rather than a thing an agent should be trusted to simulate.

`agents/` holds a small interface manifest so the skill presents correctly outside Claude Code:

```
interface:
  display_name: "Suede Deslop — AI Pattern Removal"
  short_description: "Strip AI writing tells before prose ships"
  default_prompt: "Use $suede-deslop to clean [text]. Run the merged kill list, ..."
policy:
  allow_implicit_invocation: true
```

Topics versus procedure

Most written guidance tells an agent what to think about. "Consider security implications. Check for edge cases. Make sure tests are adequate." An agent handed that produces a review that mentions security, edge cases, and tests, and none of those mentions are grounded in anything.

The dissection above never does that. Every section either names a specific pattern to look for, defines what evidence satisfies a claim, or fixes the shape of the output. Instant-F names the exact injection pattern. Grade caps name the exact sentence that lifts the cap. Output Format leaves no room to write a paragraph of impressions instead of seven letters and a gate.

A defined output artifact is the load-bearing part. Once the skill has to fill in `Checked:` and `Not checked:`, the agent cannot quietly skip a step, because the skipped step has to be written down in the second field.

The standard

Here is the test to hold your own skills to. Hand the `SKILL.md` and the target to a competent stranger with no context about you, your repo, or your habits. They should be able to run it and produce the same artifact you would.

If they would need to ask you a question first, that question is a missing section. If they would produce a different shape of output, the output format is underspecified. If they would reach a different verdict on the same evidence, the rubric is doing less than it appears to.

That standard is harsh, and it is also the whole reason the format works. A skill that passes it is a specialist you can hire an infinite number of times.

> **THE MOVE**
>
> Take your best skill, delete every sentence that tells the agent what to consider, and check whether an output artifact is still defined at the end. If not, write the output block first and rebuild the body around it.

PART I · CHAPTER 3

The Description Contract

The description field is the router. Trigger words, NOT FOR boundaries, overlapping descriptions, and what a dead route costs inside your agent.

Five skills in this pack mention code in their first sentence. A review skill, a grader, a combined pass, a CI gate, and a fifty-agent shipping DAG. You type "can you look at this PR," and exactly one of them is correct. Nothing in the system errors if the wrong one runs. You get output either way, and the output looks fine.

That is the whole problem with descriptions. They read like marketing copy and they behave like routing tables. The body of a skill can be a thousand lines of careful procedure, and none of it matters if the router never selects it. Progressive disclosure means the description is the only thing the agent reliably sees for every installed skill, every time. It is the contract. The body is the payload.

So write it as a contract.

Write the words a user would actually type

Open `skills/suede-full-send/SKILL.md` and read the description before the body. It does not describe a philosophy of maximum effort. It lists literal strings: `full send`, `max effort`, `max agents`, `spare no compute`, `throw tokens at it`, `burn tokens`, `burn max tokens`, `never end your allocation above zero`. Those are not synonyms chosen for elegance. They are the phrases a tired founder types at 1am, transcribed into the frontmatter so the router has something to match against.

This is the part most people get backwards. You know what your skill does, so you describe it in your vocabulary. The user does not have your vocabulary. They have a problem and a keyboard. If your grading skill's description says "holistic readiness assessment" and the user types "just give me an A-F, no line-by-line," the match is weaker than it should be, and a neighboring skill with sloppier prose but better word overlap takes the request.

The pack enforces this mechanically. `tests/fixtures/trigger-routing.json` defines groups of competing skills, and each route carries `positive` signals, `negative` signals, and a `metadataMustContain` list that has to appear in the skill's actual description. For `suede-code-grader`, `metadataMustContain` is `["a f", "grade"]`. If someone rewrites that description into something more dignified and drops the literal "A-F," `node scripts/validate-trigger-routing.mjs` fails. The vocabulary is a tested interface, not a style choice.

There is also an ambiguous case in that fixture, and it is the honest one:

```
{ "id": "code-ambiguous-fallback",
  "mode": "ambiguous",
  "prompt": "Check this PR and tell me if it ships.",
  "expected": "suede-code" }
```

Some requests genuinely do not disambiguate. When that happens you want a declared fallback rather than a coin flip. The group names one: `suede-code`, the combined pass, because doing both findings and a grade is the least regrettable answer to an unclear question.

NOT FOR is the other half of the contract

Forty-eight of the 73 public skills carry a `NOT FOR:` clause in their description. It always has the same shape: the condition under which a sibling wins, then the sibling's exact name in parentheses.

`suede-ship` refuses three specific neighbors: high-volume work that splits into independent worker tasks goes to `suede-codex-fleet`, findings-only review with no code change goes to `suede-code-review`, and CI and branch-protection wiring goes to `suede-ci-gate`. Read that as a routing table written from the point of view of whichever skill is most likely to steal the request.

The generic version is worthless. "NOT FOR: other tasks" tells the router nothing, because the router's question is never "is this skill relevant." Every description in a coherent pack is relevant to something. The question is "is this skill more relevant than that one," and only a named comparison answers it.

Two rules make the clause work. Name the sibling, not the category: `use suede-code-review` routes, "use a review tool" does not. Name the condition, not the vibe: "findings-only review with no code change" is testable against a request string, "smaller jobs" is not. And write it from the near-miss. Nobody confuses a shipping pipeline with a rights audit. The dangerous neighbor is always the one that does almost the same thing.

The cleanest example in the repo is a mutual pair. `amazon-returns-recovery` declines bank chargebacks, marketplace seller claims, and subscriptions billed outside Amazon, and hands them to `subscription-recovery`. `subscription-recovery` declines Amazon returns, restocking fees, and Amazon-billed Prime Video Channels, Audible, and Kindle Unlimited, and hands them back. Two skills that would otherwise fight over every refund request instead partition the space by billing relationship. Neither description needs to be cleverer than the other, because the boundary is written down on both sides.

What overlap actually costs you

When two descriptions overlap and neither carries a boundary, the agent still picks. It picks by vibes: whichever description sounds more enthusiastic about the request. The tiebreaker is prose style. That is not a metaphor for how the selection works, it is a fair description of what you are left with once semantic relevance is a tie.

Two costs follow, and they are asymmetric.

A description that oversells gets the skill loaded for work it cannot do. The skill fires, the body loads, the agent starts executing a procedure built for a different shape of problem, and it produces something. `suede-ship` running against a one-line typo fix will happily scout, research across multiple lenses, plan lanes, and gate the release. You burn real time and real context, and at the end you have a correct one-line fix wrapped in ceremony. The failure is expensive and invisible, because the output is fine.

A description that undersells never fires at all. This one is cheaper per incident and worse in aggregate, because you never learn about it. The skill sits in the pack, correct and unreachable, and you go on solving the problem by hand while believing you have tooling for it. If you have ever written a skill and then noticed six weeks later that it has never once been selected, the description undersold. Nothing will tell you. There is no error for a skill that did not run.

`suede-codex-fleet` is the case where oversell would be more than wasteful. Its description says the workers are always `codex exec` processes billed to the user's OpenAI subscription, that Claude subagent fan-out is never a substitute on any model, and that the correct behavior when the Codex CLI is missing is to halt rather than fall back. The body repeats it in a callout: the word "Fable" in the brand name is not the model `claude-fable-5`. That redundancy exists because the substitution is semantically reasonable and financially wrong. Both routes spawn parallel workers. Only one of them costs nothing against your Anthropic allocation. A description that said "parallel worker fleet for bulk generation" and stopped there would be accurate and would still let the expensive interpretation through.

Estate lint: routes that point at nothing

A `NOT FOR` clause names a skill. If that skill does not exist, you have written a dead link inside your agent's routing logic, and the agent will try to follow it.

This is not hypothetical in a pack that has both public and private halves. My personal `~/.claude/skills/` directory contains `suede-debug`, `suede-verify`, `suede-plan`, `suede-spec`, and about fifteen more that are not in the public repo. Their boundaries are real to me: `suede-debug` declines proving a finished feature works and sends it to `suede-verify`, and declines reviewing a diff for quality and sends it to `suede-code-review`. Two of those three exist for me. One of them exists for you.

So `scripts/validate-skill-pack.mjs` lints the estate. It parses every `NOT FOR` clause, collects each `(use X)` target, and fails the build when `X` is not a folder in the pack:

```
NOT FOR redirect in <skill>: named skill <name> does not exist in skill pack
```

It also keeps a `knownPrivateSkills` list, filtered down to the ones that are not public, and refuses to let any of those names appear in a public skill's frontmatter at all. Bodies may mention them, but only as a disclosed dead end, tagged with a phrase marking them unavailable. The comment in the source says why plainly: an undisclosed pointer "sends a public installer after a skill they do not have and cannot get, with nothing marking it as unavailable." Two lines in `suede-codex-fleet` were exactly that, because an earlier version of the guard read only frontmatter.

The lint itself had a hole worth knowing about. The original regex anchored every match on the literal string `NOT FOR:`, so in a clause naming three targets, only the first was ever checked. Sixteen of the twenty-one skills in the pack at that time had two or more targets on one line. Nothing broken shipped through it, but two of every three redirect targets in a three-target clause were validated by nobody. The current version finds the clause span first, then collects every `(use X)` inside it. That fix is preserved verbatim as the worked example in `skills/suede-code/SKILL.md`, which is a nice property: the pack's review skill demonstrates itself by finding a real bug in the pack's release gate.

A pack is a namespace

Ten skills have forty-five possible pairwise collisions. Seventy-three have more than two thousand six hundred. You cannot write your way out of that with better positive descriptions, because none of the descriptions are inaccurate. They are all true. Truth about what a skill does says nothing about whether it should beat its neighbor.

Which is why a set of skills is a namespace, and namespaces need the same discipline as any other: unique names, declared boundaries, no dangling references, and a checker that fails the build when someone breaks one. The alternative is not chaos. The alternative is a system that quietly does the second-best thing forever and never tells you.

> **THE MOVE**
>
> Take a request that should go to a neighboring skill, hand it to this one, and see whether the description alone sends it away. If it does not, the boundary is not written yet, and it is not the router's fault.

PART II

The Operating System

Outcome-bound work, agent lanes and worker fleets,
and the verification discipline underneath both.

PART II · CHAPTER 4

Full Send

Outcome-bound work: freeze the mission before you mutate anything, fill useful non-colliding lanes, and bring back a decision instead of a workshop.

Type "full send, fix everything" into a capable agent with no policy behind it and watch what happens. It spawns twelve subagents. Eight of them read the same four files. Three write overlapping summaries of the same finding. One produces a document titled "Comprehensive Analysis" that restates your request in headings. Forty minutes later you get a wall of process narration, a list of things that were considered, and no change on disk.

That is process theater, and it is the predictable response to maximum-effort language, because "max effort" reads as an instruction about volume. The agent has no way to distinguish between doing more work and looking like it did more work, so it does the thing that is easier to demonstrate.

`suede-full-send` exists to convert that phrase into an outcome. It is worth being precise about what it is: a policy router. Its own body says so directly. It does not change model limits, expose hidden reasoning, create external spending authority, or replace the selected controller's workflow. It never does the work. It decides who does, under what constraints, and what counts as done.

The house line, and what it does not mean

"Never end your allocation above zero."

The skill's instruction about its own catchphrase is to use it once, dryly, then return to business. It is a joke about already-authorized host compute: you are paying for the seat, the included capacity is there, so use it when using it buys something.

It is not a token counter. An agent cannot see a hidden host meter and cannot control one, and the description says so in the frontmatter so the router carries the disclaimer even when the body never loads. It is not permission to pad output. And "authorized host compute" specifically means the included in-session model and agent capacity under the host's existing controls. It does not cover separately billed APIs, credit purchases, quota increases, or cloud jobs. The mission record sets `incremental_external_spend_cap=0` and keeps it at zero until you name both a category and a maximum amount.

So the line means: do not leave useful parallel capacity idle out of politeness. It does not mean: spend more.

Bring the user a decision, not a workshop

The standard section opens with that sentence, and everything else in the skill is downstream of it. Your attention is the scarce resource. Compute is not.

Which means the default is to decide. Routine, reversible, in-scope decisions get executed, not returned to you as homework. The skill names four conditions that justify interrupting you, and the discipline is that a question must meet one of them:

1. the answer changes the desired outcome;
2. it grants new authority;
3. it crosses a serious risk boundary;
4. it chooses between materially different irreversible results.

"Should I name the variable `userId` or `accountId`" meets none of them. "Should this migration drop the old column or leave it" meets the fourth. The test is mechanical enough to apply while working, which is the point. Most agents ask too much, not because they are cautious, but because asking is cheaper than deciding and it looks like diligence.

The reporting rule follows the same logic. Report technical detail as consequence, proof, and next move. Keep progress updates brief. Do not narrate token use, lane chatter, or methodology unless asked. Nobody wants the diary.

Freeze the mission before you mutate anything

This is the single highest-leverage habit in the book, and it takes about ninety seconds.

Before any mutation, the controller writes a `FULL_SEND_MISSION` record. The full schema is in `skills/suede-full-send/SKILL.md`; the fields that earn their place are these:

- `objective`: one user-visible outcome. Not a task list.
- `targets`: exact repos, folders, routes, URLs, or accounts.
- `authorized_actions`: actions tied to those exact targets.
- `protected_wip`: dirty files, branches, and people not to disturb.
- `source_truth`: current files, live surfaces, or platform records.
- `done_signals`: commands, readbacks, URLs, or platform states.
- `risk_halts`: data loss, security, privacy, legal, payment, irreversible impact.

Every one of those is a failure you have already had. `protected_wip` is the answer to an agent cutting a branch on top of another session's uncommitted work on a shared machine. `source_truth` is the answer to an agent confidently editing a local mirror that is forty commits behind origin. `done_signals`

is the answer to "I've completed the implementation" with no command output behind it. `authorized_actions` is the answer to scope creep, which in an agent does not look like ambition, it looks like helpfulness.

The word "everything" gets an explicit definition: every required surface for the stated objective. Safe read-only discovery can add candidate surfaces to look at. It does not silently expand mutation authority. And the escalation phrases are bounded in the same way. "Full send," "fix everything," and "do not stop" increase persistence and coverage. They do not authorize purchases, deletion, publishing, third-party messages, credential handling, access changes, or irreversible external actions that were not already in scope.

Freezing the mission costs a minute and converts a vague instruction into something you can hold the run accountable to afterward. Skip it and you have no basis for saying whether the run succeeded, because you never wrote down what success was.

One controller, and lanes that actually differ

The skill selects exactly one controller. Broad work with multiple judgment, implementation, or verification lanes goes to `suede-agent-teams`. High-volume independent units that need worker briefs and review go to `suede-codex-fleet`. One contained outcome with no useful split goes to the smallest relevant specialist. If a batch is one lane inside a bigger release job, agent-teams stays the controller and the fleet is subordinate. Never assign the same units to both.

The prohibition is blunt: do not run two controllers, two plans, or two progress stores for one mission. Two plans is not redundancy. It is two sources of truth about what is done, which is worse than one wrong one.

Within the chosen controller, the instruction is to fill every useful non-colliding lane, then refill capacity while independent work remains. The qualifier is doing the work. Each lane needs a bounded artifact that can change a done signal, a decision, a risk, a surface map, or the critical path. If a lane cannot change any of those, it is not a lane, it is a witness.

So the skill rejects duplicate prose, ceremonial votes, filler agents, and semantically identical lanes, and states the principle in five words: negative evidence is useful, more words are not. A lane that reports "checked the auth flow against the OWASP list, found nothing" changed your risk picture. A second lane summarizing the first one changed nothing and cost the same.

There is one more constraint that people trip over immediately after they discover parallelism, and it is stated in the frontmatter so the router carries it: explicit Full Send intent on atomic work selects one specialist and no parallel lanes. Saying "full send" over a single-file fix does not buy you a fleet. It buys you the right specialist and stronger reasoning on the calls that matter. Parallelism is a property of the work, not of your enthusiasm. Work that does not split does not split faster with more agents watching.

Provisional until proved

Every worker result is provisional. The controller inspects the actual artifact, diff, command output, or live behavior and marks it `accepted`, `rejected`, or `fix brief`. A worker's final message never closes the mission, which is the correct default given that a subagent reporting success is a claim, not evidence.

The proof standard is specific by surface. Changed code means inspecting the diff and running the relevant build, test, lint, or focused behavior check. A skill or plugin means validating manifests, discovery metadata, install paths, and a fresh invocation. MCP means exercising JSON-RPC initialization plus the current tools, resources, and prompts. A public page means the built or live URL at the relevant desktop and mobile states. A deployment means the exact production domain and route.

Then three verdicts, and they are allowed to be uncomfortable. `PROVED` means direct evidence matches the done signal. `UNPROVED` means the signal is unchecked or supported only indirectly. `BLOCKED` means access, authority, data, or external state prevented the check. Missing proof narrows the final claim; it does not erase separately completed work. That last clause matters, because without it the incentive is to quietly claim the unproved thing rather than publish a mixed result.

The run closes with a compact brief: outcome, decision, executed, proof, adversarial reconciliation, unproved or blocked, source state, handoff, next move, status. For one atomic job, at most three sentences. For broad work, at most eight bullets. Explicitly no lane chatter, no token counts, no padded logs, no diary of the process. If the run is blocked, you get only the blocker schema plus whatever independent work actually finished, which is the version of bad news you can act on.

Effort is not the deliverable

Be honest about the failure mode this skill is fighting. Max-effort language is an invitation to theater, and a substantial part of the skill's body is suppression: no duplicate lanes, no ceremonial votes, no filler agents, no narration, no hype, no costumed role-play. A router that only added capacity would make the problem worse, because the thing that goes wrong at scale is not insufficient work, it is work that is indistinguishable from work.

Effort is an input. You cannot spend it and you cannot ship it. What you can ship is a changed surface with a command output behind it, or a named blocker with the smallest external action that would clear it. Everything between those two outcomes is cost.

C-tier answers "full send" with volume. S-tier answers it with a frozen mission, one controller, lanes that could each change the verdict, and a three-sentence brief that tells you what is true now.

THE MOVE

Before any broad run, write the seven mission fields (objective, targets, authorized actions, protected WIP, source truth, done signals, risk halts) in plain text and paste them into the prompt. If you cannot fill in `done_signals`, you are not ready to start.

PART II · CHAPTER 5

Lanes, Fleets, and Collisions

Multi-agent work that is not theater: disjoint file ownership, isolated worktrees, atomic leases, and the measured cost of a fan-out that should have been one agent.

Three agents, one repo, one afternoon. Agent A refactors the auth middleware. Agent B is told to "clean up the route handlers," which happens to include the file A is holding. Agent C runs a formatter across the tree. You come back to a branch that compiles, passes nothing, and contains two half-applied versions of the same idea. Nobody lied to you. Every agent did the work it was given.

That is the failure mode of parallel agent work, and it is not exotic. It is the default outcome when the only coordination mechanism is that the tasks sounded different when you wrote them.

What actually breaks

Four things, and they are boring, which is why they keep happening.

Two agents editing the same file. The second write wins, or the merge produces a file that satisfies neither intent. The tell in a plan is the phrase "the lanes probably won't touch the same files." `suede-agent-teams` lists that sentence in its Red Flags section, with the correct response: probably is not a lane map.

A stale mirror. Your local `main` is four days behind `origin/main`. An agent cuts a branch from it, builds cleanly against a version of the codebase that no longer exists, and opens a PR full of conflicts against code it never read. Nothing in the agent's transcript will reveal this. It fetched nothing, so it saw nothing wrong.

Someone's uncommitted WIP, destroyed. A shared checkout usually has dirty files in it. An agent that runs a checkout, a stash, a reset, or a formatter across a dirty tree can remove work that exists nowhere else. This is the one irreversible item on the list. `suede-agent-teams` puts collision detection before any lane opens: run `git -C <repo> diff --name-only HEAD` for dirty files, `git -C <repo> status --short` for untracked ones, then flag a collision if any path appears in two lane scopes, or in the dirty list plus any lane scope.

Merged-looking branches that read as unmerged. Squash merges rewrite history. After a squash, `git log origin/main..HEAD` still shows your commits ahead and `git diff` still shows changed files, because the content landed under a different hash and `main` moved on. An agent doing cleanup by ancestry will conclude the branch is unmerged and either keep a dead worktree forever or, worse, "restore" work that is already live. `git cherry origin/main` answers the real question: a leading `-` means an equivalent patch is upstream.

The fix is ownership, not politeness

Coordination between agents does not work by asking them to be careful. It works by making the collision impossible to express.

File ownership is the primitive. Every lane gets an explicit list of files it may touch, written before any builder starts, and no builder opens a file outside its list. When two lanes want the same file, `suede-agent-teams` gives you three resolutions and no fourth: independent changes get sequenced, with the second lane rebasing on the first lane's commit; overlapping changes get merged into one lane with one owner; a dirty file in a lane scope goes to the orchestrator, who either stashes and restores or makes that lane the only one allowed near it.

Notice what is missing. There is no "both lanes edit carefully" option. Splitting responsibility for a single file across two concurrent builders is not a coordination problem to solve, it is a thing you do not do.

Isolated worktrees make ownership physical rather than advisory. A worktree per lane, cut from `origin/main` rather than from your local mirror, means an agent cannot reach another lane's files even by accident, and cannot touch your dirty main checkout at all. `suede-ship` treats a violation of this as a halt condition: it stops on a lane collision or on a file held by a live sibling worktree, and its own documentation says the fix is a re-plan, not a retry. An orchestrator that retries a collision is just rolling the dice again.

The third mechanism matters when the work is a queue rather than a plan. In the public contribution program that ships with `suede-agent-teams`, tasks come from repository issues, and the duplicate-work gate is an atomic lease taken before any worktree exists:

```
node skills/suede-agent-teams/scripts/contribution-ledger.mjs claim \
  --ledger <control-dir>/contributions.json \
  --id <task-id> --worker <lane-id> --lease-minutes 45
```

Leases expire and return to the queue, long work heartbeats, abandoned work gets released. The ledger's lock has no force option. If a writer crashed, recovery requires the exact recorded token, a locally verifiable dead process, and a minimum stale age, because the alternative is two workers convinced they both own issue 123. Hard-linked ledgers fail closed, for the same reason: two names for one file can split an atomic rename into two divergent queues.

Then there is the part that decides whether any of this produced value. The handoff will not close without evidence. The Handoff Quality Checklist requires the exact repo and branch (not "the main app"), every file path changed (not "various files"), every command run with its actual output or exit code, observable verification such as a test output or curl response (not "it works"), a status from the defined vocabulary, the single most important unresolved step, and every caveat, none omitted to make the handoff look cleaner. A handoff missing a field is not done. Its status is `held`.

That checklist is what converts a fan-out from a burst of activity into something you can act on the next morning.

The economics of a fleet

Lanes are about safety. Fleets are about money.

`suede-codex-fleet` splits a bulk job into independent worker-sized tasks, writes one self-contained brief per task in `briefs/`, and spawns one `codex exec` process per brief. Claude decomposes, briefs, reviews, and assembles. The workers are OpenAI Codex CLI processes, billed to the user's ChatGPT subscription. Claude's role is admiral only.

The skill is unusually blunt about the one substitution you must not make: workers are `codex exec` processes, always, and you never swap in `Agent`, `Task`, `Workflow`, or subagent fan-out on any model, not "just for this one batch." The reasoning is arithmetic, not preference. Someone asking for a codex fleet is deliberately routing volume off the Anthropic meter. Satisfying that request with Claude models spends the exact budget the request existed to protect. If the Codex CLI is missing, not logged in, or not on `PATH`, the skill halts and asks, with an estimate of what a Claude fan-out would consume. It never falls back silently, because a silent fallback is a spending decision made on your behalf without telling you.

There is a measured number attached. On 2026-07-27 a Claude-model fleet ran in place of an explicitly requested codex fleet: 3,258 turns, roughly 1.29 billion tokens, about $1,843 of API-equivalent spend, 23% of one weekly allocation, for work that should have cost nothing on that account.

The interesting detail is where the tokens went. Ninety-seven percent of that 1.29 billion were cache reads, from workers each hauling around 500k tokens of context per turn. That is the shape of long fan-out cost. It is not the tokens the workers write. It is the same context, dragged through every turn, multiplied by workers, multiplied by turns. A worker with a fat context is not slightly more expensive than a lean one. It is a multiplier applied to the entire run.

Which gives you the operating rule: keep per-worker context small. A Codex worker never sees the Claude conversation anyway, so each brief is self-contained by construction, and self-contained turns out to mean minimal. Job, input file paths, exact deliverable, acceptance criteria the worker can mechanically self-check, and the exact output path in `out/`. The shared voice, hard bans, and output conventions live once in the workspace `AGENTS.md`, which Codex auto-loads, so briefs stay short. That file is described in the skill as the highest-leverage file in the system, and the reason is that it is the one piece of context you pay for once instead of per worker.

When not to fan out

Fan-out has a fixed cost you pay before any work happens: decomposition, briefs, lane maps, collision detection, and a review pass over every output. That overhead is worth it only under specific conditions.

Do not fan out judgment-dense work. The core principle in `suede-codex-fleet` is that Claude tokens buy judgment and Codex tokens buy volume. Clear spec plus high volume goes to the fleet. Fuzzy spec, or expensive-if-wrong, stays with the model doing the thinking. Ten landing page headlines where the whole difficulty is taste is not a fleet job, it is one writer with context.

Do not fan out one complex change across coupled files. If the files move together, they cannot be disjoint lanes, and forcing a split produces exactly the collision the lane map was supposed to prevent. That is the boundary between the two skills: `suede-agent-teams` coordinates multiple lanes on one complex change with gates and handoffs; the fleet exists for genuinely independent units. Anything with a shared manifest, a generated index, or a lockfile keeps one owner regardless of how many workers are running.

Do not fan out under a handful of items. Three briefs, three spawns, and three review passes cost more of your attention than doing three things in sequence. The overhead is real and it is front-loaded.

And do not fan out to feel productive. Twelve workers running is not twelve units of progress, it is twelve outputs you now have to read.

The actual test

Here is the question that separates a fleet from a token bonfire. For each worker's output, can the reviewer state, in advance, the criterion that decides whether it passes?

If yes, review is mechanical and the fan-out scales. The brief template forces this: acceptance criteria are numbered and mechanically checkable, limits, bans, required elements. The worker self-checks and reports pass or fail per criterion, and the skill is explicit that worker self-checks are evidence, not verdicts. A Claude review gate reads every file in `out/` regardless. Failing one or two criteria earns a one-line delta via `codex exec resume <session-id>`, not a regeneration, because regenerating throws away everything that was already right. Failing three or more, or violating an `AGENTS.md` hard ban, earns a respawn with the delta appended.

If no, if you cannot say what "good" looks like before the output arrives, you have not decomposed a job. You have distributed a vague feeling to twelve processes and committed yourself to reading whatever comes back. The volume will be impressive. The reviewer will be you, at midnight, deciding whether things look about right.

Acceptance criteria are not paperwork. They are the only thing standing between a fleet and a very expensive way to generate material nobody will check.

THE MOVE

Before spawning any parallel work, write the lane map and the per-lane acceptance criterion first, and if you cannot write the criterion for a lane, do that lane yourself instead of dispatching it.

PART II · CHAPTER 6

Evidence or It Did Not Ship

The verification discipline the rest of the book rests on, built from three real failures: a truncated test log, a page that passed a colour check while rendering unreadable, and a test file serving traffic in production.

An agent finishes a task and reports: "Fixed the failing test. The suite is green." It ran the suite. It read the output. It is not lying. What it did was pipe the run through `head`, see a wall of passing lines, and stop reading before the line that said FAILED.

That happened in this repo. It is in the memory file, under `never-truncate-test-output-before-claiming-green`. The claim was confident, the work looked like the work, and the conclusion was wrong by one scrolled screen.

This is the core failure mode of agent-assisted building, and it is not a model defect. Humans do it constantly. You report done based on the shape of the work rather than its result. You wrote the migration, so the data must be migrated. You changed the color token, so the page must be readable. You deployed, so it must be live. Each of those is a substitution of intent for observation, and each one holds until someone actually looks.

Shape is not result

The tell is easy to spot once you know it. Listen for completion claims whose support is a description of an action rather than an observation of a state.

"Updated the config so the redirect works." Did you request the URL?

"Added error handling for the webhook." Did you replay a webhook?

"The build should pass now." Should is not a verb that belongs in a status report.

`suede-agent-teams` handles this with a status vocabulary that has no shortcuts: `scoped`, `planned`, `executing`, `changed locally`, `verified locally`, `reviewed`, `committed`, `pushed`, `deployed`, `verified live`, `released`. The two adjacent pairs are where every optimistic report dies. Changed locally is not verified locally. Deployed is not verified live. The vocabulary exists because the gap between those states is invisible in a transcript and expensive in production. Its Red Flags section names the sentence that skips them: "Mark it done, the code is written."

`suede-ship` enforces the same boundary on itself. It reads production and never deploys, so its instructions forbid it from claiming `deployed`, `verified live`, or `released` under any circumstances. Those states require a deploy that has not happened. A skill that will not overclaim on its own behalf is

the only kind whose reports are worth anything.

What counts

Evidence is an artifact produced by the system, not by you, that would have looked different if the work had failed. That is the whole definition, and it disqualifies most of what gets pasted into status updates.

Counts:

- Untruncated command output, read to the end.
- An HTTP status code from a request against the live URL.
- A diff, showing what actually changed rather than what was intended.
- A screenshot of the rendered surface.
- A test run you read to the last line, including the summary.
- Row counts before and after a migration.

Does not count:

- A plan. A plan is a prediction.
- A description of a change. That is a restatement of intent.
- A partial log, or anything downstream of `head`, `grep -v`, or `2>/dev/null`.
- A build that "succeeded locally" for a claim about production.
- A worker's self-assessment. `suede-codex-fleet` puts this precisely: worker self-checks are evidence, not verdicts. The reviewer still reads the file.

The handoff checklist in `suede-agent-teams` encodes the same line. Verification means a screenshot URL, test output, curl response, or build log. It explicitly does not mean "it works." Commands are recorded with actual output or exit code, in order. A handoff missing a field is status `held`, not done, and the writer signs off on each item.

Three traps that survive careful people

The general rule is easy. The specific failures are the ones that keep landing, because each looks like verification while measuring the wrong thing.

Truncation. `npm test | head -50` looks like reading the output. It is reading the beginning of the output, which for most test runners is the part where everything is fine. Failures cluster at the end, in the summary, after the stack traces. If you are going to filter, filter toward the failure: read the tail, or grep for the failure token and the exit code, and check `$?`. A pipeline that ends in `head` cannot produce a green verdict, only a green-looking prefix.

Markup instead of rendering. A file-wide check counted the right color tokens across the docs surface and passed. Thirty-eight skills still rendered gold text on a gold background, unreadable, because the count was file-wide while the failure was per-container. The lesson recorded in this repo's memory as `verify-rendering-not-just-markup` is that a check scoped more broadly than the failure will pass through the failure. The fix is to scope the check to the container that actually renders, and to measure the computed value: contrast ratio, not token name. The related note, `docs-palette-splits-fill-red-from-text-red`, exists for the same reason. The palette's `#8b1a1a` is fills and borders only, text uses different reds, and a validator that treated the palette as one flat list would approve unreadable text.

File tree instead of production. This one is the most expensive, because the gap is not cosmetic. On some hosts, every `.js` or `.ts` file under an `api/` directory becomes a public serverless function. A file named `check.test.js` in that directory is not a test. It is a live unauthenticated route. In `suede-geo`, `GET https://scan.suedeai.ai/api/check.test` returned `200` with the body `partial`, a fixture from a mock server inside the test file. Requesting the URL executed the test suite inside a production function, booted the test's HTTP server, and hung past 25 seconds against a 300-second ceiling. Anyone could loop it.

Reading the repository would not have found that. The file looked exactly like a test, because it was one. Only the production surface disagreed. The check that works is one line per file:

```
curl -s -o /dev/null -w '%{http_code}' https://<domain>/api/<basename>
```

Anything that is not a real endpoint should return 404. Verify against the thing users can reach, not the thing you can read.

The pattern across all three is the same. Verification failed not because nobody checked, but because the check was one layer away from the claim. Tokens instead of contrast. Prefix instead of summary. Source tree instead of served route. When you design a check, ask which layer the user experiences, and measure that one.

The gate that never blocks you

There is a version of this discipline that becomes its own problem: the tool that decides it knows better and refuses to proceed. Every grading and gating skill in this repo carries the same policy block at the top, and it is worth reading as a design decision rather than a disclaimer.

Gates are advisory. Every check, quality gate, and ship verdict is a recommendation to the user, not a control on the agent. The skills run every check and report results honestly, and verdicts (`ship`, `ship-with-caveats`, `hold`, letter grades, blocked items) are advice attached to the work rather than orders

that change it. They never block, delay, skip, rewrite, or refuse the action you asked for because a check failed. The line that carries the whole policy is five words long: a failed gate changes what you report, never what you do.

One exception, and it is narrow. Extremely risky findings, data loss, security or credential exposure, legal or rights violations, payment mistakes, irreversible public damage, cause a pause: state the risk, state the options, let the user pick. Their choice is final. `suede-ship` applies the same carve-out to live production exposure it observed independently of the change at hand, such as a real secret or an unauthenticated `200` that should not exist. That goes to the user immediately, because it was true before you arrived and will be true after you leave.

This is what makes the discipline usable. A gate that blocks you becomes something you route around, and a routed-around gate reports nothing at all. A gate that reports honestly and gets out of the way stays in the loop, which is the only position from which it can tell you anything. You keep authority over the ship decision. The gate keeps authority over the description of what you are shipping.

That division is also why grading skills can afford to be blunt. There is no cost to an accurate D grade when the grade cannot stop you. The moment a verdict acquires the power to block, it acquires an incentive to be diplomatic, and a diplomatic verdict is worth nothing.

The rule

Verification is not a phase at the end of the work. It is the thing that distinguishes a claim from a guess, and a builder who ships from one who announces.

The S-tier ladder turns on it. C-tier produces output. B-tier produces working output. A-tier produces verified output. S-tier builds systems that keep producing verified output without them standing there. Every rung above C is a verification rung, which is why this is the spine of the book rather than a chapter in it.

So here is the operator's version, the one worth keeping when the rest of this fades.

You are allowed to be wrong. Wrong is recoverable, often cheap, and sometimes the fastest route to right. What you are not allowed to be is unverified and confident. That combination is the one that puts a test suite on a public URL, ships gold text on a gold background to thirty-eight pages, and reports a green suite that failed. It costs more than being wrong, because nobody goes looking for it.

Say what you checked. Say what you did not. Then the confidence you do express means something.

THE MOVE

For every completion claim you make or receive, name the artifact behind it, the command output, the status code, the diff, the screenshot, and if there is no artifact, downgrade the claim to "changed, not verified" before anyone else has to.

PART III

The Craft Lanes

Grading code, treating design and copy as systems, distribution, and the last mile.

PART III · CHAPTER 7

Grading Your Own Work

Seven scored lanes, instant-F triggers, grade caps on auth and payment surfaces, and the actual skill: predicting the grade before you run it.

A findings list is easy to ignore. You ask an agent to review a diff, it comes back with fourteen items ranked P0 through P3, you read the P0, decide it is really more of a P2, and merge. Nothing in the output stopped you, because nothing in the output was a decision. It was a menu.

A letter grade is harder to ignore. `suede-code-grader` ends with one character and a gate that follows it mechanically: A maps to `ship`, B maps to `ship-with-caveats`, and C, D, and F all map to `hold`. There is no arithmetic left to do at the end and no room to negotiate with yourself about severity. You either accept a C and hold, or you say out loud that you are shipping a C.

That is the whole trick. The grade does not add information the findings lacked. It removes the option of reading the findings and doing nothing.

Seven lanes, scored separately

The grader scores each of these A through F, then gives one overall:

1. **Correctness.** Intended behavior, edge cases, error paths, async behavior, routing, data flow, regression risk.
2. **Security and permissions.** Auth, secrets, payment, wallet, injection, path traversal, SSRF, permission and data exposure risks fail closed.
3. **Data and state.** Schemas, migrations, caches, jobs, queues, webhooks, retries, idempotency, state transitions.
4. **Suede truth.** Public copy, rights, provenance, royalty routing, licensing, product claims match the implementation. Grading work outside Suede, you substitute domain truth: API contract truth, published-statement accuracy, data model truth.
5. **UX and release behavior.** Loading, empty, error, and success states, mobile and native surfaces, metadata, routes.
6. **Tests and verification.** Changed behavior has meaningful tests, builds, screenshots, simulator runs, live readbacks, or named caveats.
7. **Deploy readiness.** Env vars, feature flags, configs, migrations, rollback notes, install paths, docs, release sequencing.

Separate lanes matter because a single number hides the shape of the risk. A change can be flawless on correctness and blind on deploy readiness, and those two failures need different responses from you. One needs a rewrite. The other needs a paragraph in the PR body and an env var pushed to production before the merge.

Notice what lane 6 does. "Tests and verification" is a graded lane, not a footnote, which means an agent that wrote perfect code and verified none of it cannot reach an A. Evidence or it did not ship, expressed as a score.

The F you cannot argue with

Before any lane is scored, the grader runs a fixed list of instant-F triggers. Six categories: secrets and credentials, injection, auth bypass, payment and wallet, data destruction, plaintext sensitive data. Two concrete examples the repo uses, because concrete is the point:

A hardcoded API key, secret, token, or password in committed source. Not in `.env`, not in a config the deploy pipeline injects, but in the actual file that went into git history. That is an F.

A role or permission check that can be bypassed by manipulating a request parameter. Someone sends `?role=admin` and the middleware believes them. That is an F.

The rule attached to those triggers is the part builders resist: **no other lane can raise a locked F.** Not a beautiful migration plan, not full test coverage everywhere else, not the fact that the key is for a staging account. The instruction to the agent is explicit: stop, report the file and line, mark the grade F, and do not grade the remaining lanes. Scoring the rest would imply those scores could matter. They cannot.

This list lives in two places, `suede-code` Step 1 and `suede-code-grader`, and both copies carry the same note: change both together. Duplicated rules drift. Saying so in the file is cheaper than discovering the drift during a review that mattered.

Caps, so a happy path cannot buy an A

Some surfaces cannot earn a top grade on passing CI alone, because CI on those surfaces usually exercises the path that was never in danger.

Auth changes (login, session, token validation, middleware, role assignment). An A requires explicit coverage of the bypass or escalation path, named: "tested with expired token returns 401", "role escalation attempt returns 403." A B requires the happy path tested plus named caveats about what is not. Meet neither and the change caps at C no matter how the other six lanes scored.

Payment and wallet flows (checkout, subscription, refund, payout, transfer, webhook). An A requires error paths tested (failed charge, declined card, webhook replay), amount and recipient validated server-side, and no silent error swallowing. Otherwise, cap at C.

Data migrations carry the harshest cap. A documented rollback plan with an untested restore gets you a B. No rollback plan at all caps the change at D, which routes to `hold` and, because data loss is one of the extreme-risk categories, puts the ship choice explicitly to you rather than letting anything proceed quietly.

Public API changes that break a contract with no migration path cap at C.

The caps exist for one reason, stated bluntly in the skill's red-flag list: "Happy path works, call it an A" is wrong because happy paths are never where the risk lives.

Grade the diff, not the story about the diff

The Source Truth section reads like a list of places to look, and it is, but the operative sentence is the first one: do not grade from the PR description or commit message alone. A PR description is a claim about a diff, written by the person or agent least able to see what the diff missed.

So the grader reads the repo, branch, remote, and dirty state; the diff, changed files, generated files, and touched routes; the imports, callers, schemas, configs, env requirements, jobs, webhooks, scripts, tests, and docs that move with the change; and whatever build, test, lint, typecheck, browser, simulator, or live evidence directly exercises the changed behavior. When live or runtime checks are not practical, it grades the source and marks those lanes unverified. Unverified is a real result. Assumed-passing is not.

The red flags that precede a fake grade are worth memorizing, because you will think all of them at some point:

- "CI passed, round up." CI that never exercised the changed behavior raises nothing.
- "The work was clearly hard." Effort never moves a grade. Evidence does.
- "It's just a refactor." Instant-F triggers run on every grade, every time.
- "The PR description is clear enough." Grade the diff and its evidence, or mark the lane unverified.

The gate is advice, and that is deliberate

Every one of these skills opens with the same policy block: the checks, the verdicts, the letter grades are recommendations to the user, not controls on the agent. A failed gate changes what the agent reports, never what it does. If you ask for a merge and the grade comes back C, you get the merge and the C, clearly labeled, side by side.

This looks like a weakness until you have used a tool that had it the other way. An agent that refuses your instruction because its own quality check disagreed has quietly promoted itself from advisor to gatekeeper, and the first thing you learn is how to phrase requests that dodge it. Then the gate is worth nothing, because it only ever fires on the honest path.

There is one exception, and it is narrow: extreme risk. Data loss, credential exposure, legal or rights violations, payment mistakes, irreversible public damage. On those the agent pauses, names the risk and the options, and hands the choice back. Your choice is final. The pause exists so that the decision is conscious, not so that it is blocked.

If you want a gate that actually blocks, that is a different skill and a different layer. `suede-ci-gate` writes the workflow and hands you the branch protection settings, and it makes the same distinction explicitly: it generates, it does not enforce. Real blocking belongs to CI and to GitHub, not to an agent in your terminal. The grader tells you the truth. Branch protection makes the truth binding.

Predict the grade before you run it

Here is the part that turns a tool into a skill.

Run the grader on your own work before anyone else does. That is the beginner version, and it is already better than most practice, because the cheapest review is the one that happens before a human's attention is spent. Ship a C knowingly if you want, but ship it knowing.

The advanced version: before you run it, write down the grade you expect. All seven lanes, plus the overall, plus which cap you think applies. Then run it and compare.

The gap between your prediction and the output is the only real measurement of your judgment. When you guess A and the grader returns C because tests and verification was thin, you learned something durable about your own blind spot. When you guess C and it returns C for exactly the reason you named, you did not need the grader for that change, which is its own useful result.

Track this for a month and the prediction converges. That convergence is the skill. The grader stops being an oracle you consult and starts being a check on a judgment you already have, which is the difference between an A-tier builder who produces verified output and an S-tier builder whose verification runs mostly in their own head, with the tool confirming rather than discovering.

A builder who cannot predict their own grade is outsourcing taste. A builder who predicts it correctly, and runs the grader anyway because prediction is not evidence, has the thing worth having.

> **THE MOVE**
>
> Before you run the grader on your next change, write your predicted overall grade and the one lane you think will be weakest in a scratch file. Run it, and keep a running log of where your prediction missed.

PART III · CHAPTER 8

Design and Copy Are Engineering

Tokens instead of hex codes chosen in the moment, numeric thresholds instead of opinions, and an anti-slop gate for prose that reads fine sentence by sentence and hollow paragraph by paragraph.

Someone picks a hex code at 1am. It looks right against the one screen they have open. Three weeks later that color exists in nine files, four of them slightly different, and nobody can say what it means. It is not the accent, it is not the warning color, it is just there, and every new component inherits the confusion.

This is the normal failure mode of design work, and it has nothing to do with taste. The person who picked that color may have excellent taste. They had no system, so their taste evaporated the moment they closed the file.

A named palette is a set of decisions with memory

This repo's own `DESIGN.md` carries eight color roles, and the roles are the point:

ROLE	VALUE
Background	`#080808`
Surface	`#111111`
Card	`#161616`
Border	`#222222`
Primary text	`#f0ece4`
Secondary text	`#888880`
Gold	`#c8a96e`
Risk red	`#8b1a1a`

Three near-blacks that differ by eight points of lightness apiece, which is what gives the surface stack its depth without a single shadow. Off-white text at `#f0ece4` rather than `#fff`, because pure white on near-black is a glare problem, not a contrast solution. Secondary text at `#888880` carrying a faint warm tint, chosen to stay AA-readable rather than to look softer.

Then the two that carry meaning. Gold at `#c8a96e` is the brand accent and the action emphasis. Risk red at `#8b1a1a` is reserved for failure, risk, and strict grading accents only. When you see red on a Suede surface, something has gone wrong or is about to. It never appears because a section needed

variety.

The rule that keeps the whole thing from collapsing sits in the spacing section, almost as an aside: **gold is used as a decision, not spread evenly across every surface.** An accent that appears everywhere stops being an accent and becomes a coating. The color is still there, the meaning is gone, and you have to invent a second accent to say what the first one used to say.

Naming does the work. `--color-risk` cannot be applied to a decorative divider without someone noticing they wrote something false. `#8b1a1a` can.

Numbers instead of opinions

"Make it more readable" is a preference. Any of the following is a check:

Body text on a dark background needs a minimum 7:1 contrast ratio, secondary text 4.5:1, disabled text 3:1. Primary actions and mobile controls provide at least a 44px touch target. Body copy stays between 65 and 75 characters per line at 1.6 to 1.7 line-height. Major type steps keep at least a 1.25 ratio between them. Cards and inputs cap at a 12 to 16px corner radius, with full-pill radius allowed for tags and buttons only. Animations move `transform` and `opacity`, never width, height, top, left, or margin. Exit curves run 220 to 280ms on `cubic-bezier(0.16, 1, 0.3, 1)`, list entrances stagger 40ms per item and clamp at six, and the whole reveal sequence caps at 480ms.

And the one that outranks the rest: **content is visible by default.** Motion may translate or soften content into place, but it must never gate visibility. A scroll-reveal that leaves text invisible when the observer does not fire is not a polish choice. It is a page that failed to render, dressed as an animation.

None of these numbers is sacred, and a different product would pick different ones. What matters is that each is falsifiable. Two people can disagree about whether a screen feels cramped forever. They cannot disagree about whether a button is 44px.

The design skill also scores whole systems on ten dimensions out of 100: color consistency, typography hierarchy, spacing rhythm, component consistency, responsive behavior, dark and light behavior, motion restraint, accessibility, information density, polish. Below 70 the system is failing, and the instruction is to fix the two lowest dimensions before styling anything new on that surface. Any single dimension at 4 or below is a P1 finding. Same principle as the code grader in the last chapter: a number forces a decision that a paragraph of notes lets you defer.

Grade the render, not the stylesheet

The most expensive habit in agent-assisted design is judging CSS by reading it. The skill's red-flag list opens with the exact thought that precedes the mistake: "The code reads right, so it will render right." Render it. Screenshots beat code inspection.

Minimum evidence is a desktop capture at 1280px and a mobile capture at 390px, via `npx playwright screenshot <url> --viewport-size=1280,900 desktop.png` or whatever your environment provides. When there is a source visual to match, the comparison happens in the same pass, never from memory, and it writes `visual-qa-report.md` with the source path, the implementation path, viewport and state, the findings by severity, and a final line that reads `passed` or `blocked`.

Two of the red flags are worth quoting because they are the ones that catch experienced people. "This change is too small for visual QA" precedes the one-line CSS change that broke mobile nav. "I remember what the reference looks like" precedes the handoff where the implementation was 20px off in a direction nobody could name afterward.

The chapter-1 rule holds here without modification. A design claim without a screenshot is a guess about pixels.

Copy has the same shape

`suede-copy` builds a page on a fixed spine: hero that names the outcome, subhead that adds audience and proof, primary CTA, proof (files, scripts, screenshots, live routes, commands), how it works in three or four steps each with a verb and a result, a safety section stating what the workflow does not claim, an FAQ, and a final CTA with less friction than the first.

Its tests are as concrete as the contrast ratios. The competitor-swap test: put a competitor's name in your headline, and if it still reads true, the headline is not specific enough. The 3-word test for CTAs: describe what happens after the click in three words, and if you cannot, the button is too vague. "Get started" fails both. "Register your first release" passes both.

Copy carries its own score, seven dimensions out of 70, with revision required below 58 and 62 required for a public launch, homepage, GitHub, App Store, or investor-adjacent surface. Same enforcement pattern, different units.

The uncomfortable part

Generated prose reads fine sentence by sentence and hollow paragraph by paragraph. That is what makes it hard to catch. Each individual line is grammatical, confident, and on topic. Read four of them together and you notice that nothing was actually claimed.

`suede-deslop` exists for that gap, and it is a merged kill list rather than a vibe check. Twenty-five highest-frequency offenders sit in a table in the skill body with a replacement for each, and forty-plus more live in `references/kill-list.md` for anything going to press, investors, or customers. Eight rules govern the pass, and a five-dimension score (directness, rhythm, trust, authenticity, density) gates the result at 35 out of 50. Below 35, revise. The red flags include the exact rationalization: "The score is 34, close enough." Below 35 means revise.

The tells, honestly listed:

Em dashes everywhere. The single highest-signal marker in current AI prose. The rule in the skill is absolute, with the anticipated objection pre-refused: "The em dash is stylistic here." No em dashes, anywhere, ever.

False agency. "The data tells us." "The decision emerges." "The culture shifts." "The market rewards provenance." Inanimate things doing human work, so that no one has to be named as the person who measured, decided, or argued. The fix is to name the actor or use "you."

Formulaic contrast. "It's not about speed. It's about precision." The binary setup announces that an insight is coming instead of delivering one, and the skill catalogs eleven separate spellings of it, from "The answer isn't X. It's Y." to "stops being X and starts being Y." Adjacent offenders: the triad rhythm ("Faster. Cleaner. Better."), the reveal fragment ("That's it. That's the thing."), and the permission grant ("And that's okay.").

Metronomic rhythm. Three consecutive sentences at the same length. Every paragraph landing on a punchy one-liner. The prose has a pulse you can set a watch to, which is what happens when nothing is being worked out on the page.

This book was written under those bans. No em dashes appear in it, the triads were cut, and every paragraph that ended too neatly got a longer sentence appended until it stopped sounding like a slogan. Whether that produced better prose is yours to judge. It did produce prose that can be checked, which is a different property from being good and a necessary one if you want the quality to survive contact with an agent, a deadline, and a bad night.

One boundary matters and the skill is strict about it: deslop edits style only. It never changes a fact, a number, a date, a name, or a price, and when a rewrite needs a specific the source did not contain, it inserts `[AUTHOR: supply X]` rather than inventing one. Style passes that quietly improve your statistics are how false claims enter a public page wearing better sentences.

The standard

Every rule in this chapter started as somebody's preference. The 44px target, the 65 to 75 character measure, the reserved red, the ban on em dashes: none of them descended from first principles. Someone made a call.

The difference is that the call got written down with a number attached, so it can be applied by a person who was not in the room, enforced by a validator, and argued with on the merits when it turns out to be wrong. That is the whole transformation, and it is available for any part of your work you currently handle by feel.

Taste you cannot write down is a preference. Taste you can write down is a system.

THE MOVE

Take the one visual or verbal judgment you make most often by feel, write it as a number or a named token in your repo's `DESIGN.md`, and apply it to the next three things you ship.

PART III · CHAPTER 9

Getting Found

Distribution as an engineering surface: the shift from ranking to being cited, nine audit lanes, an A-F visibility grade, and why a page that cannot be quoted will not be cited.

A great product with no distribution surface is a private hobby. That is not a moral judgment, it is an accounting one. You spent the compute, you passed the gates, you shipped verified output, and the number of people who can act on it is zero. The work is real. The reach is not.

Distribution used to be somebody else's department. You built, marketing promoted, and the handoff was a Slack message with a link in it. That arrangement broke when the thing doing the finding stopped being a person scanning ten blue links and started being a model assembling an answer. The model does not scan. It extracts. And what it can extract from your page is decided by choices you make in the artifact, which puts distribution squarely back on your side of the line.

What actually changed

Ranking in a results page and being cited in an answer are different jobs with different failure modes. Ranking rewards a page that a crawler can reach and a human can be persuaded by. Citation rewards a page that a machine can lift a paragraph out of without inventing anything.

Four things change in the artifact when you optimize for the second one.

Claims carry sources. A page that says "faster than the alternatives" gives a model nothing to quote, because quoting it would be quoting an opinion with a confident font. A page that says "the pack ships 73 skill folders, MIT licensed" and links the repo gives the model a fact with a provenance trail attached.

Structured data validates. Not "we added JSON-LD." Validates, at schema.org/validator, with a type and property combination that is actually eligible for the feature you think you are earning. Invalid schema is worse than no schema, because it looks like a signal and behaves like noise.

Headings answer questions. "Getting started" is a filing label. "How to install the pack in three commands" is an answer fragment a model can quote directly and attribute to you. `suede-visibility-grader` states the spread plainly in its AI readability sub-rubric: "Getting started" = F, "How to install X in 3 commands" = A.

Paragraphs stand alone. If the third paragraph of your docs page only makes sense after reading the first two, an extractor will either take all three or take none. Write the unit you want quoted.

The nine lanes

`suede-seo-audit` runs the audit as lanes, each scored A through F, each with a checklist that lives in the skill's `references/` folder so the body stays routable. Its core principle is one line: never audit from memory. Every finding cites what was actually fetched.

Lane 0 is keyword research, optional, informational only, and it produces a brief rather than a grade. Numeric demand stays `unknown` unless a dated source provides it, which is the skill declining to invent traffic estimates to make a finding land harder.

Lane 1 is technical access. Status code, final URL after redirects, canonical, `robots.txt`, sitemap presence, whether primary content survives without JavaScript. The lane drops to C or below on an actual indexability block, a wrong canonical, a failed redirect, or missing primary content. Using JavaScript is not a failure. Core Web Vitals get measured when tooling allows and never enter the grade.

Lane 2 is search and answer intent. One coherent intent, one primary reader, one earned action, no cannibalization against your own other pages. The lane drops to C or below when the opening section answers a different question than the URL implies.

Lane 3 is metadata. Title, description, Open Graph, Twitter card, alt text, author entity. Character counts are preview diagnostics, not limits, because Google publishes no fixed limit and rewrites titles anyway. The lane drops to D or below when the page's entity cannot be identified from title, description, and H1 together. Length alone never causes a drop.

Lane 4 is structure. Heading hierarchy, section order that follows reader intent, descriptive anchor text. No universal template, no word count quota, no keyword density target.

Lane 5 is schema. Decide whether structured data is warranted at all, then validate every block and confirm it matches visible content. When schema is missing or broken, the skill requires the literal corrected JSON-LD inline in the findings. Describing the fix in prose is explicitly not allowed. The lane also carries an honest caveat most SEO advice skips: FAQ markup earns no visibility promise, because Google generally limits FAQ rich results to authoritative government and health sites.

Lane 6 is AI EO, answer engine optimization. Clear opening answer, plain definitions, explicit subjects, accessible primary content, source links, flagged hallucination risk. It also records that Google Search ignores `llms.txt`, and tells you not to grade its presence as a search signal. That is the lane refusing to sell you a ritual.

Lane 7 is copy and conversion quality. Lane 8 is E-E-A-T, treated as a human trust lens rather than a ranking score Google exposes, graded on how many of the four have real visible evidence. Lane 9 is topic cluster architecture, skipped for single-page audits and marked N/A when a pillar model does not serve the site.

The grade is mechanical, derived from finding counts and nothing else. A means no HIGH or MEDIUM findings in the lane. F means four or more HIGH findings, or the lane is absent or actively harmful. Lanes 1, 3, and 6 count 1.5x. Overall A requires 38 of 42 weighted points, and is capped outright if you never verified a live URL, if the primary CTA is broken, or if any published statement is false, unverifiable, or invented.

The blunt grade

The full audit is a deliverable. Sometimes you want a verdict. That is `suede-visibility-grader`, which answers one question:

```
Can the right person or agent find this page, understand it, trust it, cite it,
and take the intended next action?
```

Six lanes, A through F: findability, first-screen clarity, CTA pull, proof and trust, AI readability, and design signal. First-screen clarity grades the rendered first viewport, not the document structure, because a visitor does not read your DOM. AI readability has that six-item sub-rubric, ending in a test worth stealing: if a model were asked "what is this?" right now, would this page produce a correct, non-hallucinated answer? If no, the lane caps at C.

The caps are where the skill earns its keep. No live inspection caps the overall at C. A broken primary CTA caps at D. A false published statement caps at D, or F if the statement is central to the product promise. Mobile not inspected blocks A outright. Quick grade mode, the fast gut-check read of the first viewport only, cannot assign A at all. Its maximum is B.

Surface type changes the standards, which is the part generic SEO advice never does. On a README, findability barely matters because GitHub handles it, but AI readability is elevated, because models cite READMEs constantly, and one broken install command caps Proof at D. On a product page, proof is the primary gate and vague next steps like "learn more" cap CTA pull at D. On a docs page, CTA pull is graded leniently, because docs exist to inform.

Proof outranks adjectives

Both audiences punish the same thing. A human reads "industry-leading" and discounts it, because everyone writes that. A model reads it and cannot cite it, because there is nothing there to attribute. `suede-site-alchemy` states the check directly: remove or rewrite "used by thousands" without a number, "industry-leading" without a comparison, "fast" without a metric.

The replacement is not better adjectives. It is a number with a source, a command with output, a screenshot of the thing working. `suede-seo-audit` has a Lane 7 failure condition that reads like a dare: the lane drops to C or below if the copy could belong to any competing product without changing a word.

The funnel side

Being found and being acted on are separate problems, and `suede-site-alchemy` handles the second. It starts by naming the funnel stage before optimizing anything, because a page that serves the wrong stage fails no matter how polished it is. Top of funnel makes the problem vivid and does not ask for commitment. Middle of funnel differentiates and handles objections. Bottom of funnel removes doubt with proof, pricing clarity, and real risk reversal.

Then the friction audit, which counts three kinds. Cognitive: how many decisions before the primary CTA, how many value propositions compete on the first screen. Physical: how many form fields before any value is delivered, how many clicks, whether a mobile visitor scrolls past anything before seeing a CTA. Trust: whether an objection goes unanswered before the ask, whether proof arrives before the request, whether the guarantee sits near the button or is buried in a footer.

The audit is explicitly an inventory, not a score. Each item records affected population, evidence, severity, and the event that would show improvement. A raw count proves nothing.

The conversion math is honest about what it is. The model reads `observed_revenue = eligible_visitors × observed_CTA_rate × observed_close_rate × observed_order_value`, and the skill labels it descriptive, not a causal forecast. Unmeasured fields stay `unknown` rather than silently absorbing an industry benchmark. If someone needs a planning range, it goes out as a sensitivity table with every assumption labeled, called a scenario, never an expected lift.

Ranked quick wins come out of that, at most three hypotheses after the friction audit, ranked by evidence strength, affected population, decision value, effort, and risk. Never by invented projected lift. Broken paths and missing conversion events get fixed before anything visual, because a beautiful page with a dead CTA is a page with a dead CTA.

There is one rule in the urgency section that generalizes past marketing: before adding a deadline, ask whether the claim would still be accurate if a visitor came back in 30 days. If not, it is a dark pattern, and you cut it.

The rule

A page that cannot be quoted will not be cited. Read your own page looking for one paragraph a stranger could paste into an argument without adding context or a caveat. If you cannot find it, the model will not either, and it will go quote someone who wrote one.

> **THE MOVE**
>
> Take your most important public page, open the first viewport only, and write down the one sentence a model would return if asked "what is this?" If that sentence is not already on the page, verbatim, put it there.

PART III · CHAPTER 10

Shipping Into the Real World

The last mile: App Store and Play Store gauntlets, release evidence gates, and a procedure that recovered $448.31 by running outside a repo entirely.

The last mile is where builders stall, and it is almost never for lack of skill. It stalls because the last mile is the first place your assumptions meet an institution that does not care about them. A compiler tells you immediately. App Review tells you in four days. Google Play tells you after the staged rollout has already reached 5% of users. The feedback loop stretches from seconds to days, and every unverified assumption you carried through the build gets billed at the new exchange rate.

You can feel this the first time you wrap a website in a shell and submit it. The build works. The app runs. Rejection arrives citing guideline 4.2, minimum functionality, and now you are two weeks out from a launch date because the thing you shipped was a bookmark with an icon.

The wrapper gate

`site-to-ios-app` puts that failure at the front instead of the end. Its principle is stated before any command: turn a site into an iOS app only when the app has native value, stable iOS behavior, and a release surface that is truthful. A raw web page in a frame is not enough for an App Store-quality product.

The workflow begins with an artifact, not a scaffold. You write `SITE_TO_IOS_AUDIT.md` capturing the site URL, target user, primary routes, login requirements, iPhone responsive behavior, PWA signals, legal/support/account-deletion links, payments and sensitive flows, session behavior, mobile performance risks, native value opportunities, and the 4.2 wrapper risk. Only then `SITE_TO_IOS_PLAN.md`, with the chosen strategy, the native value to add before release, scaffold and build commands, bundle ID and signing notes, QA matrix, screenshots and metadata and privacy work, blockers, and the explicit release gate.

The strategy decision is four options and a requirement to write down why. Capacitor remote shell keeps the live site as the product surface, so web deploys update most behavior. Capacitor bundled shell packages static assets into the binary, which means updates need an App Store release unless paired with live APIs. A native SwiftUI shell wraps a web view in real navigation, settings, auth, push, share, and error surfaces. Full native rebuild is for sites that are mostly content, have weak mobile UX, or carry high wrapper rejection risk.

The 4.2 gate itself is the honest part. Block or redesign the app when it is only a bookmark, content mirror, or unmodified website. The remedies are specific: iOS-native onboarding, empty states, errors, offline, and retry. Native settings with support, privacy, terms, account deletion, restore, and

notification controls. Universal links. Share sheet, widgets, push, media pickers, StoreKit, but only when they serve the app. Safe-area, keyboard, navigation, dark and light mode, and dynamic type handling.

Then a completion bar that does not let you round up. The project builds on a named simulator, device, or CI target. Every plugin and entitlement is justified by actual behavior. The 4.2 risk has a mitigation. Screenshots and metadata match implemented features. Privacy answers match the actual SDKs, cookies, analytics, and account flows. No secrets or signing material are committed. Submission itself requires the user to explicitly delegate public release and confirm the exact app, bundle ID, version, build, and account.

The Android version of the same lesson

`android-app-factory` runs the same discipline against a platform with more moving policy. Its principle: build the product, policy evidence, and Play listing together, because a successful release is an installable app whose claims, disclosures, entitlements, privacy behavior, and store configuration agree with each other.

It refuses to answer policy from memory. Every release-oriented run identifies the repo, application ID, branch, Play app, target track, and whether the checkout is dirty, reads a dated policy baseline, then re-opens the linked official Google sources for anything submission-sensitive and records the URL, observed requirement, and check time. The baseline is treated as a baseline, not as permanent law. The skill even states its own dates carefully: as of 2026-07-19 the default is `compileSdk` and `targetSdk` 36, with Play's announced enforcement for new apps and updates moving to API 36 on 2026-08-31 and API 35 enforced until then, plus an instruction not to misstate an announced deadline as an enforced one.

The pipeline is ten phases. Validate the product. Verify policy. Design architecture and risk with the smallest maintainable shape that preserves unidirectional state, lifecycle safety, offline and error and loading states, and test seams. Scaffold native Kotlin and Jetpack Compose by default, resolving current stable versions from official sources into a version catalog, and prove a debug build before feature work. Build one complete core loop. Prove quality across unit, repository, ViewModel, Compose UI, accessibility, and end-to-end tests, plus static analysis, release build, device and API matrix, baseline profile, and Macrobenchmark. Add Play Billing for covered digital goods, processing pending purchases, verifying and acknowledging after entitlement handling, restoring ownership, and keeping server verification off the device. Complete trust surfaces: privacy policy, Data Safety, permission rationale, in-app and web account deletion when accounts exist, content rating, ads declarations, app access instructions. Build store artifacts. Release through evidence gates.

The release gate is a template you copy into the app repo and complete with links or command output. It blocks on a list that reads like a catalog of things people assume: policy not checked live, a build targeting the wrong API, an Android 15+ release not verified for 16 KB page-size compatibility including transitive native SDKs, failing tests or lint or `bundleRelease`, a core loop that fails on the declared

matrix or offline path, launch-critical accessibility failures under TalkBack or large text, missing performance evidence, disagreement between Data Safety and actual behavior, unproven billing restore or acknowledgement, committed secrets, listing screenshots showing behavior the release build does not have, or a rollout without explicit user confirmation.

One line in there is the whole chapter compressed: a caveat must have an owner, a risk, and a next action, and policy, security, privacy, billing, crash, and core-task blockers cannot be downgraded to cosmetic caveats. That is the sentence that stops a release checklist from becoming a formality.

Packaging is an artifact

`suede-launch-packaging` handles the surface between finished work and a stranger. Its core principle: a release nobody can install is not a launch. Nothing is live until you fetched it yourself, and no install path ships until the exact command ran from a clean temporary directory.

That second clause is the one that catches people. Your install command works because your local checkout has files you never pushed. The skill's failure table names it directly: "works locally, fails for a public user" usually means the command references a local plugin alias, and "install command 404s" usually means the skill folder was never pushed to `main`. The fix is not to read the command more carefully. It is to run it somewhere that has none of your context.

The skill starts with an inventory, because one launch is usually several surfaces. A skill launch is a repo plus a README plus an install command plus social copy, and each row needs its own proof artifact: repo URL and commit hash, live URL and screenshot, install command transcript, MCP `tools/list` output, live route readback, link sweep results. Its hard gates are ordering rules. No launch copy until the live surface is verified. No install doc until the command ran from a clean temp dir after pushing. Ship gate set before any announcement, including a soft one.

The lane that proves the point

The sharpest example in the pack does not touch a repo at all.

`amazon-returns-recovery` started as a test of a general-purpose contract-negotiation procedure. The question was whether the discipline held up in a dispute. Pointed at a live Amazon account, it surfaced two restocking fees of $44.99 and $28.50 on returns processed weeks earlier, charges the account owner did not know existed. Both were waived in one sitting. The first refunded exactly $44.99, the second came back at $30.63 because the associate rounded up past the fee. Then a third case outside the skill's default scope: a $372.69 electric shaver on which a refund had already been denied once, while still inside the return window. Re-disputed two days after the window closed, it ended in a full $372.69 refund with no return required. Total across the documented cases: $448.31.

Notice what carried over. Not domain knowledge about Amazon. The procedure knew nothing about restocking fees before it went looking. What carried over was the shape: read the evidence first, itemize before arguing, build the case from facts you can point at, ask for one specific outcome, confirm the result in writing. That is the same shape as a code review, a release gate, and an SEO audit. A procedure that works outside its home domain is evidence that the procedure, not the domain knowledge, was doing the work.

`subscription-recovery` is the generalization, pointed at App Store, Google Play, PayPal, bank statements, and direct-bill services. It exists because somebody noticed the discipline had nothing to do with Amazon.

Safety, stated plainly

Both recovery skills draw the same line, and it is worth reading as written rather than paraphrased. Read-only discovery first. Report the findings as a plain list. Stop. Do not open the dispute chat yet. Confirmation before any cancellation or refund contact, one item at a time. The skill describes itself as a metal detector, not an autopilot.

Truthfulness is a hard constraint, not a preference. State only true facts: order number, item, price, fee amount. Do not invent a prior contact attempt or a return reason that did not happen, because the ask is reasonable on its own and does not need embellishment to land. Accept one polite counteroffer round at most, then report back rather than escalating with anything untrue. Some fees are legitimate, some subscriptions are still wanted, and the skill instructs you to say so when a finding looks earned rather than mistaken.

Credentials are never handled. The stretch goal for automated statement parsing is explicitly deferred because no validated path exists that avoids touching credentials, which conflicts with the skill's own rules. A capability was left unbuilt rather than built unsafely, and that decision is written down in the skill where the next person will read it.

The reframe

Shipping is not the phase after building. It is a separate skill with its own gates, its own evidence requirements, and its own failure modes, and you get better at it the same way you got better at everything else: by doing it enough times that the checklist stops surprising you. The builders who stall at the last mile are not worse engineers. They have just practiced the first mile a thousand times and the last mile twice.

THE MOVE

Take the thing you shipped most recently and run its install command from a fresh temporary directory on a machine that has never seen the project. Whatever breaks is what your users have been hitting.

PART IV

Becoming S-Tier

The ladder, the judgment layer, authoring your own skills, and ninety days of practice.

PART IV · CHAPTER 11

The S-Tier Ladder

Five tiers defined by observable behavior, with the week, the transcript fingerprint, the failure that holds you there, and the move that promotes you. Most readers place themselves one rung high.

Most builders rate themselves by how much they produce. That is the wrong axis, and it is why the ladder in this chapter will feel unflattering. The tiers here are defined by what someone can prove about their output and by what keeps working when they close the laptop. Volume does not appear anywhere.

The ladder is not a personality test. It is not about ambition, work ethic, years of experience, or how much you care. Two builders with identical attitudes land two tiers apart because one of them runs a test suite before saying "done" and the other says "done" and waits to hear otherwise. The difference is observable in a transcript. That is deliberate: every rung below is described by behavior you can go check, in your own repo, this afternoon.

Agents make this ladder sharper than it used to be. When output was slow, tier differences showed up as throughput and everyone could pretend the gap was time. Now anyone can generate a thousand lines in four minutes, so throughput tells you almost nothing. What survives as a signal is verification, scope control, and whether the same problem has to be solved again next month.

D: produces output

A D-tier builder ships text that looks like a solution. The agent wrote it, it reads confidently, it went into the repo. Whether it works is an open question that someone else usually answers.

Their week is a sequence of things that were "done" and then were not. Monday's feature comes back Wednesday because it never handled the empty state. A migration lands and a report breaks two days later, discovered by a user. Their calendar fills with rework they do not label as rework, because each incident feels like a fresh bug.

Their transcripts have a shape. Long agent responses, then "great, next," then a new task. There is almost never a command output in the transcript. No `npm test` block, no `curl` response, no screenshot, no diff read back. When something fails, the next message is a paste of the error and the word "fix." That loop can run five times on one bug without anyone stating what the bug actually is.

The specific failure: they cannot tell working from broken without an external signal. Not "they do not bother." They lack a mechanism. If nobody complains, it worked. This means their confidence and their correctness are uncorrelated, and the uncorrelation is invisible from the inside.

The move that promotes them is small and boring. Before you say a task is finished, paste one piece of evidence into the thread: a test run, a request and its response, a screenshot of the actual rendered page, a diff you read. One artifact. Not a plan to add tests later. The rule is the one this book keeps returning to: evidence or it did not ship. Applying it once per task, mechanically, is the entire move.

C: produces working output on familiar ground

C-tier is genuinely competent inside a known perimeter. They know their stack, their repo, the three services they touch every day. Inside that box the work is correct and they can tell you why.

Outside the box, everything resets. A new language, an unfamiliar deploy target, a subsystem they inherited, and they are back to guessing. Not because they are incapable, but because their method is recall, not investigation. When recall returns nothing, there is no fallback procedure.

Their week is bimodal. Familiar tickets close fast and cleanly. Then one unfamiliar item sits open for three days and closes with a change nobody can fully explain. Someone asks why it works and the honest answer is "that combination stopped the error."

The transcripts show it. On familiar work: short, precise instructions, good results. On unfamiliar work: the prompts get longer and vaguer, the agent starts proposing options, the builder picks one on vibes, it half works, and the next twenty messages are variations of the same fix. There is no point where anyone writes down what they believe is true and then checks it.

The specific failure: no transferable method for unknown territory. Familiar competence is being mistaken for competence.

The move is to run one unfamiliar problem through an actual procedure instead of intuition. `suede-debug` exists for exactly this, and the substance of it fits in a sentence: state the hypothesis, name the observation that would disprove it, run that observation, keep or discard. The point is not the skill file. The point is that when you leave known ground you switch from recall to method, on purpose, and you can feel the switch happen.

B: produces working output reliably, and knows why it works

B-tier is where most strong builders live, including a lot of people who would place themselves at the top of this ladder. Their code works. They can explain the mechanism. They handle unfamiliar problems with real method. They are the person a team escalates to.

They are also the bottleneck for everything they touch, and this is the part that stings, because it does not feel like a deficiency. It feels like being important.

Their week is full. Their own tasks, plus reviewing everyone else's, plus the three questions per day that only they can answer because the answer lives in their head. They work late not on the hard part but on the accumulated small parts. When they take a week off, throughput does not drop by one person's

worth. It drops by more, and things wait.

Their transcripts are good. Clear briefs, real constraints, evidence checked. And every single one starts from scratch. The same twelve lines of context about how this repo handles auth get retyped, in slightly different words, in forty different sessions. The same review checklist gets described from memory each time and is slightly different each time. They have a method and it lives nowhere but in them.

The specific failure: everything they know is stored in a person rather than in an artifact. That is not a knowledge problem, it is a distribution problem, and it caps them permanently, because a person does not scale and does not run in parallel.

The move: take the thing you explained three times this month and write it down where the agent will load it. A `CLAUDE.md` entry, a skill file, a checked-in checklist. One artifact per repeated explanation. The test of whether you did it right is that the next time the situation comes up, you do not have to be present.

A: produces verified output, with evidence, on unfamiliar ground

A-tier is the first rung where the builder's own claims can be trusted without a second opinion. They work in unfamiliar territory with method, they produce output that runs, and they attach the proof before anyone asks for it.

The distinguishing behavior is self-review that is actually adversarial. Not rereading their work approvingly. Deliberately looking for the thing that would embarrass them, with the same rigor they would apply to a stranger's pull request. An A-tier builder is the one who says "this passes CI but CI never exercised the changed path, so treat the test lane as unverified." Nobody made them say that.

Their week has fewer surprises than a B-tier week, and the surprises that do arrive are new rather than repeats. Their reviews come back with small notes because the large notes were already found and fixed. When they report a status, the status holds. That last property is worth more than it sounds like: an organization can plan against an A-tier builder's word.

Their transcripts show separation between building and checking. There is a point in the session where the tone changes and the builder starts trying to break their own work: running the error path, hitting the endpoint with a bad token, checking the mobile viewport, reading the migration for the rollback story. `suede-code-grader` encodes the standard they are holding themselves to, and its red flags read like a list of A-tier temptations resisted. "CI passed, round up." "The work was clearly hard." "It's just a refactor." "Happy path works, call it an A." Effort never moves a grade; evidence does.

The specific failure that holds them at A: every unit of verified output still requires them to be in the loop for it. They have made themselves reliable and they have made themselves necessary. The A-tier ceiling is a throughput ceiling wearing a quality costume.

The move: pick the check you perform most often by hand and make it run without you. The review pass becomes a required CI check. The deploy verification becomes a script with a readback. The context you re-explain becomes a skill description precise enough that the agent loads it unprompted. One check, moved out of your hands, permanently.

S: builds systems that produce verified output without them

S-tier is defined by a property of their absence. When they are not in the session, the work still comes out verified. Not because the agent is smarter, but because the standard lives in files the agent loads.

Their week looks strange from outside. Fewer tasks appear on their name. More of their hours go into things with no user-visible output that week: a skill description rewritten so it routes correctly, a grade cap added for payment surfaces, a handoff template that will not close without evidence, a CI gate that turns a soft opinion into a merge blocker. Then a task they never touched ships correctly and the reason is traceable to one of those files.

Their transcripts have a distinctive fingerprint: short prompts and long, correct outputs. A B-tier builder writes 400 words of context per session. An S-tier builder writes 30, because the other 370 are already loaded from a skill body that the description routed to. Progressive disclosure is doing the work. The prompt is small because the system is large.

The second half of the definition is the harder half, and it is where most people who reach for S-tier fall over: knowing which problems deserve a system and which deserve to be done once and forgotten. A system has a cost. It has to be written, maintained, and kept honest as the repo drifts underneath it. A skill that encodes a stale convention is worse than no skill, because the agent will follow it confidently.

The specific failure at S: systematizing the wrong things. Building a framework for a problem that occurred twice and will not occur again. Writing a generalized abstraction after seeing one instance. Producing tooling that has to be maintained forever to serve a case that lasted a quarter. This failure feels exactly like good engineering while you are doing it, which is why it is dangerous.

The heuristic that keeps it honest is frequency times consequence. If a task recurs often and being wrong about it is expensive, it deserves a system. If it recurs often and being wrong is cheap, it deserves a checklist. If it happened once, do it and move on. The instant-F list in `suede-code-grader` is a good example of the first category: hardcoded secrets and auth bypasses are both recurrent and catastrophic, so they get an unconditional mechanical rule that no other lane can override. UI spacing magic numbers appear in the same skill and are explicitly given no grade impact. Same author, same file, two different judgments about what deserves machinery.

The move at S is not "build more systems." It is to look at your last five systems and ask which one you would delete. Then delete it.

Place yourself honestly

Five questions. Answer them about the last two weeks, not about your intentions.

1. In your last five completed tasks, did every one of them include a command output, URL, screenshot, or diff you actually read before you called it done?
2. When you last worked in an unfamiliar area, did you write down a hypothesis and the observation that would disprove it, or did you try things until the error stopped?
3. Is there anything you explained more than twice this month that is still not written down where an agent would load it?
4. If you took two weeks off, name the specific thing that would break. Can you point at the file that would prevent it?
5. Of the systems you built in the last quarter, which one has been used by someone other than you, on a task you did not supervise?

Read the answers this way. Any "no" on question one holds you at D regardless of everything else. A "tried things until it stopped" on question two holds you at C. A "yes" on question three holds you at B, no matter how good your verification is. If question four has no file, you are A at best. Question five is the only one that distinguishes S from a very organized A, and if the honest answer is "nobody, yet," you are A. That is not an insult. A is a good place to be and most people reading this are not there.

Most readers will place themselves one rung high, and the correction is usually question three. Knowing something and having written it down feel similar from the inside and are not remotely the same.

The trap at the top

S-tier is not about producing more. It is about where the marginal hour goes.

A B-tier builder with a free hour does another task. An S-tier builder with a free hour improves the thing that does the tasks. Both hours feel productive. Only one of them changes next month's capacity, and only one of them is a bet, because the systematizing hour pays nothing today and might pay nothing ever.

That is the honest cost. Every hour spent on the system is an hour of visible output not produced, and there is no guarantee of a return. A skill that nobody invokes is worse than nothing: it is maintenance debt with a helpful name. The `suede-recommend-next-action` scoring rubric names this directly, and it is worth borrowing outside the skill. Leverage scores two points only when the work "fits one focused session and prevents rework or creates a reusable result," and zero when it is "unscoped, multi-day, or low-payoff." Both halves matter. Reusable is not enough if it does not fit in a session, and bounded is not enough if nothing reuses it.

The failure mode at the top of this ladder is not laziness. It is a builder who has learned that systems are the answer and stops asking whether this particular problem is a system-shaped problem. They end up maintaining an estate of machinery that serves a workload that moved on. From inside, that feels like S-tier. From outside, it looks like someone very busy with their own tools.

The tier that actually matters is not a rank. It is a question you can ask about any hour you just spent: if this exact situation shows up again in six weeks, will it cost the same, or less? D through B pay full price every time and get faster at paying it. A pays full price but never pays twice for the same mistake. S has arranged for someone else to not pay at all, and can tell you which bills were worth arranging and which ones they should have just paid.

THE MOVE

At the end of this week, find the one thing you did more than twice and would do again, and spend an hour turning it into a file the agent loads instead of a paragraph you retype. Then check next week whether anything actually loaded it.

PART IV · CHAPTER 12

Taste, Judgment, and Knowing When to Stop

The layer no procedure supplies: which decisions are yours, why a bug fix that becomes a refactor is a judgment failure, and how to tell finished from merely different.

An agent will keep going forever. Not metaphorically. Give it a repo and an open-ended instruction and it will find work in that repo until you interrupt it, and every piece of that work will look defensible in isolation. This is the single most important operational fact about working with agents, and almost nothing in the tooling protects you from it. Stopping is the operator's job. There is no procedure that supplies it.

Everything in this book up to now has been a system: a skill, a gate, a rubric, a proof standard. This chapter is about the layer above all of that, the part that decides which system to run, how far to let it go, and when the thing in front of you is finished. That layer cannot be written into a SKILL.md, because writing it down is exactly the move that fails. A rule that says "stop when it is good enough" is not a rule.

What can be transmitted is a set of decision boundaries, a few tests, and a practice for accumulating judgment on purpose instead of by accident.

Which decisions are yours

The first judgment is about judgment itself: which calls belong to the agent and which come back to you.

Getting this wrong in one direction produces an agent that asks permission to rename a variable. Getting it wrong in the other direction produces an agent that force-pushed to main because it seemed cleaner. Both failures come from the same missing rule, and `suede-full-send` states it in four clauses. Bring the user a decision, not a workshop. Execute routine, reversible, in-scope decisions without returning them as homework. Ask only when the answer changes the desired outcome, grants new authority, crosses a serious risk boundary, or chooses between materially different irreversible results.

Those four conditions are worth taking apart, because each one is doing distinct work.

Changes the outcome means the answer alters what gets delivered, not how it gets delivered. Which of two column names to use does not change the outcome. Whether the export includes deleted records does.

Grants new authority means the agent is about to reach a surface it was not given. Reading a file it was pointed at is in scope. Reading a different repo to get context is a new surface, and it does not become authorized because it would be helpful.

Crosses a serious risk boundary is the short list that never gets relaxed: data loss, credential or privacy exposure, legal or rights claims, payment, irreversible public action. The house line of that skill, "never end your allocation above zero," is a dry joke about using already-authorized compute hard. It has never meant spending money or touching production because persistence was requested. Full send increases persistence and coverage. It does not create authority.

Materially different irreversible results is the subtle one. Two options, both defensible, both permanent. Postgres or SQLite for a project that will run for years. A public API shape that other people will build against. Neither is wrong, and the wrongness only appears in eighteen months, and by then the choice is not revisitable at reasonable cost. That is your decision, always, and an agent proposing one confidently is not evidence that it is right.

The corollary is stricter than most people apply it: everything outside those four clauses, the agent decides. If you find yourself approving choices that fail all four tests, you have not been careful, you have been a bottleneck with a nice explanation.

Scope discipline, or how a bug fix eats a Thursday

A bug fix turns into a refactor roughly like this. The agent opens the file with the bug. The file is bad, and the agent notices, correctly. The fix would be cleaner with the function split. Splitting the function means updating four callers. Two of the callers have their own problems. Ninety minutes later the pull request is eleven files, the original two-line fix is in there somewhere, and the reviewer cannot separate the fix from the churn.

Every step of that was locally reasonable. That is what makes it dangerous.

Treat it as a judgment failure, not a productivity win, and be specific about why. The reviewer can no longer verify the fix, because verifying it now means verifying eleven files. The blast radius went from one function to four call sites, so the risk of the change no longer matches the risk of the bug. Rollback got worse: reverting the fix now reverts the refactor. And the refactor never got the design attention it would have gotten as its own piece of work, because it arrived as a side effect.

The clean version is unglamorous. Fix the bug in the smallest diff that fixes it. Write down the refactor you did not do, in an issue or a TODO with enough context to act on. Ship the fix. Decide about the refactor separately, when you are deciding about refactors rather than when you are fixing a bug.

There is a hidden benefit. Most refactors written down that way are never done, and a meaningful fraction of those should never have been done. The urge to clean up a file is strongest while you are inside it and weakest when you are looking at your priorities. The second view is the more accurate

one.

`suede-agent-teams` builds this into its lane map: every lane declares the files it owns, and no builder opens a file outside its assignment. That is scope discipline made structural, so it stops depending on anyone's restraint in the moment. When a lane genuinely needs a file it does not own, that surfaces as a collision the orchestrator resolves, which is exactly right. Scope expansion should be a decision someone makes, not something that happens.

Second-best now versus best later

There is a class of decision where the options are close and the deliberation is expensive. Which HTTP client. Which of two acceptable schemas. Whether to use the framework's router or a small custom one for six routes.

For these, the second-best option chosen in ten minutes beats the best option chosen in three days, and the margin is usually not close. The best-option analysis costs three days of not shipping, and it produces a decision that is maybe five percent better on a dimension you have not measured, based on a usage pattern you are guessing at because the product does not exist yet.

The reason this is safe is reversibility. If you can change the HTTP client in an afternoon, the cost of being wrong is an afternoon, which means the deliberation should never cost more than an afternoon. Time spent deciding is bounded by the cost of being wrong, not by how interesting the decision is.

Then there is the other class, where this advice is actively harmful. Public API shapes. Data models that will accumulate years of rows. Auth architecture. Anything other people will build against. Here the second-best option chosen fast becomes a permanent tax, and the deliberation is cheap by comparison, and you should take the three days.

The question is not "which option is better." It is "what does being wrong cost, and when do I find out." Cheap and soon: decide in ten minutes. Expensive and late: this is one of the four clauses from earlier, it is your call, and it deserves the strongest reasoning you have.

Agents make the first class faster and the second class more dangerous. Faster because they will just implement the reversible thing and you can look at it running. More dangerous because an agent will produce a confident, complete, internally consistent design for an irreversible decision in ninety seconds, and confident and complete is exactly what a bad architectural decision looks like before it is deployed.

Deciding what not to build

The strongest judgment move available is declining, and it produces nothing you can point at, which is why it is undersupplied.

Three questions do most of the work.

Who asked, and what did they actually want? A feature request is usually a proposed solution wearing the clothes of a problem. Someone asks for CSV export because they need one number in a spreadsheet once a month. The export is a week. The number is an endpoint and an afternoon. Answer the underlying need, not the proposed shape of it.

What does it cost after it ships? The build is the small number. The support burden, the migrations it constrains, the tests it adds to every future change, the fact that it must keep working forever: that is the real price. Anything you ship, you own. A feature used by two percent of users still blocks a schema change three years later.

What does it prevent? Not just the time. The next thing you would have built, and the fact that the product now means something slightly different than it did, and every future decision has to accommodate the new surface.

The default should be no. Not out of scarcity, out of arithmetic: the number of things worth building is much smaller than the number of things worth considering, and shipping is the expensive part.

The done test

Here is the honest test for whether something is finished: are you still improving the artifact, or are you now only changing it?

Improving means a specific defect closes. A bug goes away, an untested path gets a test, a confusing name gets clear, a missing error state appears. You can name what was wrong before.

Changing means the fourth revision of a paragraph that was fine in revision two, the button moved four pixels and then back, the abstraction reshuffled because a different arrangement is also nice. Nothing was wrong. Something is now different.

The tell is that you can no longer state the defect you are fixing. If the honest description of your next edit is "I want to see how it looks the other way," you finished a while ago and have been redecorating since. Ask what defect this edit closes, and if there is not one, stop.

The second test is harder and better: what evidence would change your mind about it being done? If you cannot name a check that would come back negative, you are not verifying, you are admiring. `suede-full-send` gives three verdicts for exactly this reason. `PROVED` means direct evidence matches the done signal. `UNPROVED` means the signal is unchecked or supported only indirectly. `BLOCKED` means something outside your control prevents the check. Notice that `UNPROVED` is a legitimate way to finish. You are allowed to ship with a named, honest gap. What you are not allowed to do is call an unproved thing proved, and you are not allowed to keep working past done because you are avoiding the discomfort of shipping something with a named gap.

The agent will never stop

None of this used to be urgent, because effort was self-limiting. You got tired. The scope of your ambition was governed by the length of your afternoon.

That governor is gone. An agent given "improve this codebase" will improve it for as long as you let it, and will produce a plausible summary at every checkpoint, and will never once volunteer that the remaining work is not worth doing. It has no concept of diminishing returns because it has no stake in the return.

This changes what your role is. In the old arrangement you supplied the effort and the environment supplied the limits. Now the agent supplies the effort and you supply the limits, and if you do not supply them nobody does. `suede-full-send` puts a hard clause on its reconciliation loop for this: repeat only while a named authorized action targets a specific unresolved signal and has a plausible material effect. Read it as a stopping condition, because that is what it is. Once you cannot name the signal, the loop is theater.

Practically: name the done signal before you start, not after. If it is not written down at the beginning, you will discover it at the end, and what you discover will be shaped by what the agent happened to produce.

Taste is written down or it does not exist

Taste sounds innate. It is not. It is a set of preferences you have named, with reasons attached, accumulated over enough decisions that they arrive fast enough to feel like instinct.

The accumulation only happens if you write down what you rejected and why. Rejection is where the information is. Everyone keeps a record of what they built. Almost nobody keeps a record of the four approaches they considered and killed, which is a shame, because the reasons those died are the actual content of judgment. "We do not use that pattern here" is a rule you cannot apply to a new situation. "We stopped using that pattern because it hides the error path and we shipped two silent failures" is a rule that generalizes to things that are not that pattern.

Two habits make this concrete. The RFC format in `suede-agent-teams` requires an Alternatives Considered section: two or three options with the reason each was not chosen. That requirement exists because the alternatives are the durable part of the document. Six months later nobody rereads the proposed solution, it is in the code. They reread why the other thing was rejected, usually because someone is proposing it again.

The second habit is per-session and takes a minute. When you reject something an agent produced, say why in one line before moving on. Not "no, try again." Say "no, this swallows the error, I want it to surface." That line is worth writing even when the agent does not need it, because you are the one

accumulating. Rejections stated as reasons become preferences. Rejections stated as vetoes evaporate.

Over a year of this you have a document nobody assigned you: a list of the things you will not do and the specific damage each one caused. That is taste, and it is portable, and it is the one thing in this book that an agent cannot generate for you because it is made entirely out of your own consequences.

Stopping

Every skill in this pack has a stopping condition. Grades cap. Loops require a named unresolved signal. Handoffs will not close without evidence. The pack is built that way because the alternative is a machine that runs until you notice, and by the time you notice it has eleven files open and a very reasonable explanation.

The discipline is simple to state and hard to hold: decide what done looks like before you start, and then honor it when you get there, including on the days when the work is going well and continuing feels better than shipping. Especially those days. Momentum is the most expensive reason to keep going, because it is the one that never announces itself as a reason.

An agent will produce work forever. Deciding that a thing is finished is the last judgment nobody can delegate, and it is the one that separates a shipped product from an impressive amount of activity.

> **THE MOVE**
>
> Before you start the next task, write the done signal in one sentence and put it where you will see it, then stop when you hit it even if the session is going well.

PART IV · CHAPTER 13

Write Your Own

Authoring skills: when to write one, what goes in the body versus references, the competent-stranger test, and how to lint an estate before a dead route bites.

The third time you paste the same release checklist into a chat window, stop typing and open a file instead.

That is the whole trigger. Not a strategy, not an assessment of whether your process is "mature enough to systematize." You have explained a procedure to an agent three times, or you keep a checklist in your head that you half-remember and half-improvise, and the parts you forget are always the same parts. Those are the two signals. Everything in `skills/` in this repo started as one of them.

The second signal is worth being precise about, because it is easier to miss. It is not "I do this often." It is "I do this often and I get it slightly wrong in a different place each time." A procedure you execute perfectly from memory does not need a file. A procedure where you remember the four steps and forget the fifth, and it is a different fifth every time, is a skill waiting to be written down. The forgetting is the value. You are not encoding what you know, you are encoding what you drop under load.

The anatomy, from the author's side

You already know the shape from the reader's side: description routes, body pays off. Writing one inverts the emphasis. Almost all of your care goes into the frontmatter, and almost all of your discipline goes into the body's output contract.

The frontmatter is two fields.

```
---
name: suede-deslop
description: "Strip AI writing patterns from prose before anything goes public. Em dashes,
filler openers, manufactured enthusiasm, false agency, passive voice, formulaic structures, all
of it. Use when copy, a README, an email, a social post, or a doc is about to ship, after a
long AI-assisted writing session, or when text sounds fine but feels generated. NOT FOR:
writing new copy (use suede-copy); changing or certifying facts, which must be checked against
primary evidence before publication."
---
```

Read that description as three parts doing three jobs. The first sentence names the durable job. The middle names the use-when conditions in the words a user would actually type: "about to ship," "sounds fine but feels generated." The last part is `NOT FOR`, and it hands the near-miss cases to a named neighbor.

Write the use-when clause in your trigger words, not your taxonomy. This is the single most common authoring failure. You know the skill as "prose quality normalization," so you write that, and then you type "this reads like ChatGPT wrote it" and nothing loads. The description is the router and the router matches on the language of the request. Before you write it, write down three to five prompts you would realistically type, verbatim, and two or three that should go somewhere else. The description has to cover the first set and the `NOT FOR` has to cover the second.

The `NOT FOR` route is not politeness. Two skills with overlapping descriptions make routing depend on loader order, which means the same prompt gets a different skill on a different machine. An explicit route makes the boundary a fact rather than a coin flip. Name the neighbor by skill name, in parentheses, and make sure that neighbor exists.

Then the body. Three things earn their place, and most bodies that fail are missing the third.

Source Truth. What the agent must read before it does anything. Not "understand the context," a list. `suede-code-grader` opens with it: read the repo, branch, remote, dirty state, then the diff and changed files, then the imports, callers, schemas, configs, and tests that move with the change, then build and test evidence that exercises the changed behavior. And a blunt line above it: do not grade from the PR description or commit message alone. That line exists because an agent will absolutely grade from the commit message if you let it.

The procedure. Numbered steps, thresholds, and commands. Narrative only where it changes a judgment call. The rule that keeps bodies honest: no naked judgment words. Every "good," "polished," "safe," "high-risk" in a skill body must resolve to a number, a checklist, or a command whose output decides it. `suede-deslop` does not say the prose should be clean; it runs a merged kill list against a 50-point score gate. If you cannot say how an agent would score the quality, the quality is decoration and you should cut it.

A defined output artifact. The step people skip. A skill that ends without saying what file, what report shape, what sections, and what fields will produce something different on every run and you will never be able to tell whether it worked. Specify the artifact. `suede-visibility-grader` has an `## Output Format` section and then a `## Sample Report` section showing the thing filled in. The sample is not padding. It is the fastest way to make a weaker model produce the right shape.

One more, and it applies to any skill that says something "blocks": say what happens at the block. Stop, name the blocker in one line, list two to four resolution options, wait. A skill that says "this blocks release" and then keeps working has written a comment, not a gate.

Bundled files, and when to split

A skill folder is a directory, and the markdown file is only one thing in it. Deterministic procedures go in `scripts/`, heavy reference material in `references/`, reusable inputs in `assets/`, `templates/`, or `fixtures/`.

The split rule is about loading cost, not tidiness. The body gets pulled into context every time the skill fires. Anything that is long, rarely consulted in full, or purely enumerative should live in `references/` and get named from the body. `suede-deslop` keeps a 226-line kill list in `skills/suede-deslop/references/kill-list.md` rather than inline, because the agent needs the method every run and the exhaustive phrase list only when it is actually scanning. `suede-agent-teams` does the same with `references/public-contribution-program.md` and keeps `scripts/contribution-ledger.mjs` as executable rather than as prose describing what the script would do.

The counter-rule matters just as much: keep it inline when splitting would hide the rule. A four-line threshold table belongs in the body. Move it to `references/` and the agent now has to decide to go look, which it often will not.

The test for a good skill

A competent stranger runs it on your work and produces the same artifact you would have produced.

Not "understands your intent." Produces the artifact. That is the bar, and it is unforgiving in a useful way, because it fails on exactly the things you cannot see in your own writing. Read your skill as if you were a small model with no context and no history with you. Every place you would have to guess, the skill loses. Add the threshold, the command, or the halt format that removes the guess.

Then run the boundary version of the test: take a prompt that should go to the neighboring skill and check that your description does not catch it. A skill that fires too often is worse than one that fires too rarely, because it displaces the correct skill silently.

Linting the estate

Once you have more than a handful, the failure mode changes. Individual skills stay fine and the connections between them rot.

Two things rot first. **Dangling routes**: a `NOT FOR: ... (use suede-foo)` where `suede-foo` was renamed, or never existed, or lives in a private repo the reader does not have. The agent reads a route to nowhere and improvises. **Missing relative assets**: the body references `scripts/check.py` and the file was never committed. Both are mechanically checkable, and the forge linter checks exactly them, exiting nonzero on any route target not present in the target surface or a declared external root, and on any relative path under `scripts/`, `references/`, `assets/`, `agents/`, `templates/`, `fixtures/`, `examples/`, or `data/` that does not exist.

The third rot is worse because it is invisible and it is not about skills at all. It is duplicated numbers. This repo says "73 skills" in the README badge URL, the README badge alt text, the README intro, five separate install sections, `PRODUCT.md`, `CITATION.cff`, `docs/llms.txt`, four different `docs/` pages' title

tags and og:descriptions and JSON-LD blocks, both plugin manifests, the MCP catalog, and inside `skills/suede-workflow-skills/SKILL.md` itself. Add a skill and every one of those is wrong, and not one of them will tell you.

So they are guarded. `scripts/validate-skill-pack.mjs` carries a `countChecks` array: file, label, regex with a capture group, expected value. The comments in it are a field log of what actually broke. The shields.io badge value drifted to 29 while the alt text and all the prose said 67. Five install sections said "all 70 skills" while the badge and intro said 71, which is why that check carries `every: true` and validates every occurrence instead of the first match. The docs catalog page had no count check at all, so its title said 28 while the meta description on the same page said 67. There is also a structural check that counts the rows the catalog page actually lists, because one lane badge read 4 above 5 rows and no string check would ever have caught it.

The lesson generalizes past this repo. Anything missing from `countChecks` goes stale silently, so when you add a surface that states a count, add its guard in the same commit. `npm test` runs the validator with `--strict` plus the trigger routing contract, the MCP protocol tests, and the Python suite.

Actually adding one

Four steps, no ceremony.

```
mkdir -p skills/suede-release-notes
$EDITOR skills/suede-release-notes/SKILL.md
```

Write the frontmatter first, in your trigger words, with the `NOT FOR` route. Then Source Truth, procedure, output artifact, boundaries, routing. Then install it. For the whole pack, `bash install.sh` rsyncs every skill folder into `~/.claude/skills/` and prints the count, leaving non-pack skills alone. For one skill you are iterating on, copy the folder into `.claude/skills/` in the project so it is scoped to the repo you are testing against:

```
mkdir -p .claude/skills
cp -R skills/suede-release-notes .claude/skills/
```

Then verify the route, which is the step everyone skips. Start a fresh session, type one of the prompts you wrote during the pressure test, and confirm the agent loads your skill rather than a neighbor or nothing. Fresh session matters: an agent already holding the procedure in context will look like the routing worked when it did not. If it does not fire, the description is wrong, not the body. Rewrite the use-when clause in the exact words you just typed.

None of this required a vendor. `skills/<name>/SKILL.md` is a text file in your repo. It diffs, it reviews, it reverts, it merges, and when the tool that reads it changes its mind about something, you edit a file instead of filing a support request. The skills in this pack are MIT licensed, which means the first thing

you should do with any of them is fork the parts that fit your work and delete the parts that do not. They are a starting shape, not a product surface. Your estate is yours.

THE MOVE

Take the procedure you have explained to an agent three times this month, write it as `skills/<name>/SKILL.md` with a description in your own trigger words and one `NOT FOR` route, then start a fresh session and confirm it loads before you write another line of the body.

PART IV · CHAPTER 14

The First Ninety Days

A real practice plan in three phases, a list of anti-goals, two honest measurements, and the close: never end your allocation above zero.

Most people install a skill pack, run two skills, feel mildly impressed, and go back to typing paragraphs at a chat window. The tooling was never the problem. The problem is that a new capability without a practice attached decays to a novelty in about a week.

So here is a practice. Ninety days, three phases, specific enough to argue with. It assumes you already have the pack installed and roughly an hour a day of real work you would be doing anyway. Nothing here is extra work. It is the same work, run through a different discipline.

Days 1 to 30: judgment

The first month is not about coverage. It is about calibration. You are training one thing: the ability to predict what a review will say before it says it.

Week 1. Install and run. `/plugin marketplace add JasonColapietro/suede-creator-skills` then `/plugin install suede-skills@suede`, or clone and `bash install.sh`. Then run `suede-code` on your own work, every day, on whatever you actually changed. Staged diff, current branch, does not matter. Read the findings. Do not fix anything yet. You are reading, not reacting.

Week 2. Predict the grade. Before you invoke the grader, write down the letter you expect and one sentence of why. Then run it. The gap between your guess and the card is the only metric that matters this month. You will be wrong in a consistent direction: most people over-grade their own work by a full letter, and almost everyone is blind to the same lane, usually data and state or deploy readiness. Find your blind lane by week 2 and you have gotten most of the value of the month.

Week 3. Evidence lines. Start writing proof into every claim you make, to yourself, to an agent, in a PR body, in Slack. Not "the endpoint works." The command you ran and what it printed, the URL you loaded and the status code, the commit hash. This will feel pedantic for about four days and then it will start catching things, because the moment you go to write the evidence line is the moment you discover you never actually checked. A claim without a command output, a URL, or a diff is a guess wearing a suit.

Week 4. Write one skill by hand. Not generated, not adapted, typed. Pick the procedure you have explained three times. `skills/<name>/SKILL.md`, frontmatter in your own trigger words, one `NOT FOR` route, Source Truth, procedure, defined output artifact. Install it, open a fresh session, confirm it

routes. One skill, written properly, teaches you more about how the whole mechanism works than reading forty of someone else's.

By day 30 you should be able to predict a ship grade within one letter, and you should have one file in your repo that you wrote and that an agent loads.

Days 31 to 60: procedure

The second month is about sequencing. Month one made you a better judge. Month two makes you a worse improviser, which is the point.

Week 5. Freeze the mission before mutating anything. For every task larger than one file, write the outcome, the boundaries, and the definition of done before a single edit lands, and do not renegotiate it mid-run. Most agent work that goes sideways goes sideways because the target moved while the work was in flight, and nobody noticed because the agent cheerfully followed the drift. A frozen mission also gives you something to check the result against that is not your memory of what you wanted an hour ago.

Week 6. Run one real multi-lane job. Real, meaning work you would have done anyway, not a demo. The non-negotiable constraint is disjoint file ownership: every lane owns a set of paths, and no two lanes own the same path. Write the ownership map before you spawn anything. If two lanes need the same file, they are one lane, or one waits. This is the entire difference between parallelism that saves time and parallelism that produces a merge conflict you spend the afternoon untangling. Keep per-worker context small; cache reads dominate cost in fan-out work, and a worker dragging half a repo through every turn is expensive in a way that does not show up until the bill.

Week 7. Build a ship gate that gates a merge. Not a workflow file that runs tests and reports. A required check on your default branch that will actually stop a merge. `suede-ci-gate` covers stack detection, one required check, and branch protection wiring. The test of whether you did it right is simple: open a PR that breaks something and confirm the merge button is unavailable. If you can still merge, you built a dashboard.

Week 8. Audit one live surface. Your README, your landing page, your docs index, whatever a stranger hits first. Run `suede-seo-audit` or `suede-visibility-grader` against it and fix what comes back. Work nobody can find is work that did not ship, and this is the week most builders discover their best project has a title tag from a template.

By day 60 you should have a mission-freeze habit, one merge that a gate blocked, and one public surface that is measurably more findable than it was.

Days 61 to 90: systems

The third month is the one that separates the ladder's top rung from the one below it. A-tier produces verified output. S-tier produces systems that keep producing verified output without them.

Week 9. Convert your three most-repeated procedures into skills. You have been noticing them since week 4. Write all three. Give each a real `NOT FOR` route to its nearest neighbor, including your own skills, so the estate routes rather than collides.

Week 10. Lint the estate. Every route target resolves. Every relative path under `scripts/`, `references/`, `assets/`, `templates/`, or `fixtures/` exists. Every number you state in more than one place has a guard that fails when it drifts. If you have a validator, add the check in the same commit as the surface that states the number. If you do not, the number will be wrong within two months and nothing will tell you.

Week 11. Hand off a whole workflow and verify the artifact, not the process. Pick something end to end, brief it, walk away, and come back to the output. Then check the output against the frozen mission and the evidence lines. Do not watch the transcript. Watching the process is how you end up with an expensive autocomplete and a tired operator, and it is a habit that quietly caps you at however many streams your attention can hold. The whole point of a defined output artifact is that it can be checked cold.

Week 12. Ship something to a real audience. Strangers, not your group chat. A repo, a post, a tool, a page. This is not a motivational flourish. It is the only test that closes the loop on the previous eleven weeks, because a real audience is the one reviewer that does not grade on your intent. Everything before this is rehearsal with a friendly judge.

Anti-goals

Things not to do in ninety days, each of which will eat the whole quarter if you let it.

1. **Do not read all 73 skills.** You will use eight. Find those eight by working, not by studying the catalog.
2. **Do not build a personal framework in month one.** You have not earned the opinions yet. Frameworks written before the practice encode your current mistakes and then defend them.
3. **Do not fan out for the sake of fan-out.** Parallel lanes on work that does not decompose produce conflicts and cost, and the cost lands on a budget you are not watching.
4. **Do not chase the grade.** A grader you are optimizing against stops being a measurement. Fix the finding, not the score.
5. **Do not automate a procedure you have not run manually three times.** You will encode the version of it that does not work.

6. **Do not skip the fresh-session routing check.** A skill you believe is installed and is not is worse than no skill, because you stop noticing that the step is missing.
7. **Do not measure this in hours saved.** See below.

Measuring the change honestly

Hours saved is the wrong number, and it is wrong in a way that will mislead you for a year. It is unfalsifiable, because the counterfactual does not exist. It rewards the wrong behavior, because generating more output faster is exactly what C-tier does. And it feels great right up until you notice that the volume went up and nothing got more reliable.

Measure two things instead. First: **what fraction of your claims can you prove?** Take the last twenty statements you made about your own work, in PRs, in updates, to yourself. How many have a command output, a URL, or a diff behind them? In month one that number is usually embarrassing. By month three it should be most of them, and the ones that cannot be proven should be labeled as guesses out loud. That single change does more for how people trust your work than any amount of throughput.

Second: **how much work continues without you?** Count the procedures that survive your absence. A gate that blocks a merge while you are asleep. A skill a collaborator invokes without asking you what it does. A validator that catches the stale number in a commit you never reviewed. Each of those is one thing that no longer depends on you remembering. Zero is where everyone starts. Three or four by day 90 is a different kind of builder than the one who started.

Both numbers have the property that you cannot fake them to yourself, which is the only reason to prefer a metric.

Never end your allocation above zero

It is the house line of `suede-full-send`, and it gets misread constantly. It is not about a token counter, and it is not permission to burn compute to look busy. It is a dry joke about effort that has already been authorized.

The situation it names is specific. Someone has told you to finish something. The budget, the access, and the mandate exist. And there is a lane still open, a check not run, a surface not verified, a claim not backed. Stopping there does not save anything. The allocation was already spent on getting to that point. The only thing left over is unfinished work, and unfinished work does not carry forward. It just sits there, slightly wrong, until someone else finds it.

So the line means: when you have been authorized to finish, finish. Fill the useful lanes, not the padding ones. Run the check that would embarrass you. Verify the live surface instead of the file tree. And then stop, because the other half of the line is that the allocation has a floor and hitting zero means done, not more. Nobody is impressed by effort spent past the outcome.

Which is the ordinary version of everything in this book. Skills are a way to write down what you already know so it survives you forgetting it. Gates are a way to make your standards hold when you are not there to enforce them. Evidence is a way to be believed by people who have no reason to take your word. Multi-lane work is a way to spend attention on decisions rather than on supervision. None of it makes you a better builder on its own. It makes the version of you that shows up tired, on a bad week, with three things overdue, produce roughly the same quality as the version that shows up sharp. That is the whole trick, and it is worth more than any individual clever hour you will ever have.

You already have the repo. Ninety days is not long. The only thing that separates the person who does this from the person who reads about it is that one of them opened a file today.

THE MOVE

Start the clock now: run `suede-code` on whatever you changed today, write down the grade you expect before you look, and put the gap in a note you will still be adding to on day 30.

APPENDIX A

The Skill Index, by Intent

Every public skill in the pack, grouped by what you are trying to do rather than by what it is called.

Every public skill in the pack, grouped by what you are trying to do rather than by what it is called. Each one lives at `skills/<name>/SKILL.md` and can be read before you install it.

Invoke a skill by naming it (`Use suede-code to review my staged diff`) or by describing the work in the words its description was written to catch.

I want the whole outcome, not a task

SKILL	USE IT WHEN
`suede-full-send`	You want a broad authorized outcome finished end to end, with one controller and proof at the close
`suede-ship`	A repo change spans several files or surfaces and should be researched, decomposed, reviewed adversarially, and release-checked in one pass
`suede-agent-teams`	Complex work needs coordinated lanes with file ownership, collision checks, rollback plans, and an evidence-backed handoff
`suede-codex-fleet`	The job is high-volume and splits cleanly into worker-sized tasks, and you want OpenAI Codex CLI workers generating while Claude reviews
`suede-recommend-next-action`	You are stalled and want one scored recommendation with a runnable prompt, not a menu
`suede-workflow-skills`	You want the umbrella that loads the pack

I want to know if this code is safe to ship

SKILL	USE IT WHEN
`suede-code`	You want findings and an A-F ship grade in one pass
`suede-code-review`	You want the bugs, not a verdict: TypeScript, React, Next.js, OWASP, accessibility, SEO, database, deploy risk
`suede-code-grader`	You want the blunt letter grade only, with instant-F triggers and auth and payment caps
`suede-ci-gate`	You want CI that actually holds the line: path-aware builds, one required check, branch protection, no deadlock
`suede-ai-eval`	The surface is an LLM, RAG, classifier, or agent, and you need rubrics, failure modes, eval cases, and acceptance gates
`suede-mcp-qa`	You ship an MCP server and want drift caught before release

I want it to look and read like someone made it

SKILL	USE IT WHEN
`johnny-suede-design`	You want the full design lane across brand and product surfaces
`suede-design`	You need tokens, color, type, hierarchy, motion, dark mode, or visual QA on shipped screens
`johnny-suede-write`	You want the full writing lane: structure, persuasion, anti-slop gate, copy score
`suede-copy`	You need conversion copy: sections, email, microcopy, buttons, headlines, variants
`suede-ship-copy`	One high-stakes public surface has to be true, and you want research lenses, a claim audit, and a publish-readiness gate
`suede-deslop`	Prose is finished and about to go public, and it still sounds generated
`suede-image`	You need marketing images: generation prompts, hero and social graphics, mockups, export sizing
`suede-video`	You need format choice, scripting, storyboarding, generation, or editing for marketing video

I want people to find it

SKILL	USE IT WHEN
`suede-seo-audit`	You want a nine-lane evidence-based SEO and generative-search audit with exact rewrite fixes
`suede-ai-seo`	You want to be cited inside AI answers: extractable structure, citable claims, `llms.txt`, agent-readable pages
`suede-visibility-grader`	You want an A-F grade on a live page for findability, clarity, CTA pull, proof, and AI citation readiness
`suede-site-alchemy`	You want the page turned into a conversion path: friction, proof, CTA, pricing, quick wins
`suede-programmatic-seo`	You are building data-backed keyword, location, directory, integration, or comparison pages at scale
`suede-content-strategy`	You need pillars, clusters, cadence, distribution, and refresh priorities
`suede-competitors`	You are writing honest alternative, versus, and comparison pages for evaluators
`suede-directory-submissions`	You are selecting listings, sequencing submissions, and verifying backlinks
`suede-free-tools`	You want engineering-as-marketing: calculators, graders, generators, scored and scoped
`suede-clip-to-guide`	A clip, talk moment, or transcript should become a funnel package with rights routing

I want to launch it

SKILL	USE IT WHEN
`suede-launch-packaging`	Finished work needs a README, docs, install commands, proof links, QA, and release copy
`site-to-ios-app`	A website, PWA, dashboard, or marketplace should become an App Store app, past the 4.2 wrapper gate
`android-app-factory`	You want a native Kotlin and Compose app taken from idea to a signed staged Play rollout
`suede-aso`	You need store keyword fields, titles, subtitles, descriptions, screenshots, and competitor listings
`suede-public-relations`	You need a media list, a validated story angle, a pitch, or a press kit
`suede-campaign-in-a-box`	A song or release needs a full rollout: hooks, rituals, visuals, merch, calendar, email, site copy

I want to grow it

Acquisition and outbound: `suede-ads`, `suede-ad-creative`, `suede-cold-email`, `suede-prospecting`, `suede-social`, `suede-instagram-growth`, `suede-sms`, `suede-co-marketing`, `suede-community-marketing`, `suede-lead-magnets`, `suede-referrals`.

Monetization: `suede-pricing`, `suede-offers`, `suede-paywalls`, `suede-signup`, `suede-onboarding`, `suede-churn-prevention`, `suede-emails`.

Measurement and operations: `suede-analytics`, `suede-attribution`, `suede-ab-testing`, `suede-revops`, `suede-sales-enablement`, `suede-marketing-plan`, `suede-marketing-loops`, `suede-marketing-ideas`, `suede-marketing-council`, `suede-marketing-psychology`, `suede-product-marketing`, `suede-customer-research`, `suede-competitor-profiling`.

Forty of these are adapted from marketingskills by Corey Haines under the MIT License. See `NOTICE.md`.

I want to protect what I made

SKILL	USE IT WHEN
`suede-rights-audit`	You need the ownership, splits, samples, provenance, and licensing gaps found before packaging
`suede-rights-passport`	You need an evidence-scoped rights handoff: works, recordings, releases, parties, identifiers, claims
`suede-release-linter`	A creative release folder needs auditing before handoff
`suede-sync-packaging`	Songs need sync-pitch prep: scene angles, one-sheets, clean and instrumental assets, mood tags

I want my money back

SKILL	USE IT WHEN
`amazon-returns-recovery`	Amazon returns, restocking fees, short or denied refunds, and forgotten Amazon-billed subscriptions need auditing and disputing
`subscription-recovery`	Recurring charges outside Amazon need finding and auditing across App Store, Google Play, PayPal, and direct-bill services

Both report findings first and require your confirmation before any cancellation or refund contact. Neither handles credentials.

APPENDIX B

The Rules, on One Page

Everything the book argues for, compressed to 29 rules across evidence, decisions, scope, parallelism, skills, craft, and stopping.

Everything the book argues for, compressed. Print it, or do not, but be able to recite the first six.

Evidence

1. A claim without command output, an HTTP status, a diff, or a rendered screenshot is a guess in a confident tone.
2. Read test output to the end. A pipe into `head` has hidden a `FAILED` line before and will again.
3. Verify the rendered result, not the markup. A file-wide count can pass while the page is unreadable.
4. Verify against production, not the file tree. What deploys is not always what you think you wrote.
5. If a step was skipped, say it was skipped. Partial work reported as complete costs more than partial work reported as partial.

Decisions

1. Bring a decision, not a workshop. Your operator's attention is the scarce resource, not compute.
2. Ask only when the answer changes the outcome, grants new authority, crosses a serious risk boundary, or picks between materially different irreversible results. Otherwise decide and report.
3. Routine, reversible, in-scope calls are yours to make. Handing them back is not caution, it is delegation upward.
4. A concern raised once and overruled is settled. Say your piece, then build the thing.

Scope

1. Freeze the mission before you mutate anything: objective, targets, authorized actions, protected work in progress, source of truth.
2. The requested scope is the deliverable. Do not quietly narrow it, widen it, or transform it into the work you would rather do.
3. A bug fix that becomes a refactor is a judgment failure with good intentions.
4. Finish the parts that are not blocked, then state plainly what you left out and why. Scaling the work down is the requester's call.

Parallelism

1. Lanes must be disjoint. Two agents editing one file is not parallelism, it is a merge conflict with extra steps.
2. Fan out for volume, never for judgment. One complex coupled change is one agent's job.
3. Every worker needs an acceptance criterion before it starts. Without one you are generating, not building.
4. Isolate with worktrees. Check for live processes before deleting one, and do not trust a session list to tell you what is running.

Skills

1. The description is the router. Write it in the words a user would actually type.
2. Every skill needs a NOT FOR route to its nearest neighbor, or the two will fight over the same request and one will win at random.
3. A skill is good when a competent stranger runs it and produces the same artifact.
4. Write one when you have explained the same procedure three times.
5. Gates advise, they do not block. A failed check changes what you report, never what you do. The single exception is extreme risk: data loss, credential exposure, rights violations, payment mistakes, irreversible public damage. There, stop and let the human choose.

Craft

1. Taste you cannot write down is a preference. Taste you can write down is a system.
2. Thresholds beat opinions. Measured contrast, 44px targets, 65 to 75 characters per line.
3. Proof outranks adjectives, for humans and for models.
4. A page that cannot be quoted will not be cited.

Stopping

1. Done is when further work changes the artifact rather than improves it.
2. An agent will happily continue forever. Stopping is your job.
3. Never end your allocation above zero. Dryly, once, then back to business.

Never end your allocation above zero.

skills.suedeai.ai/book/