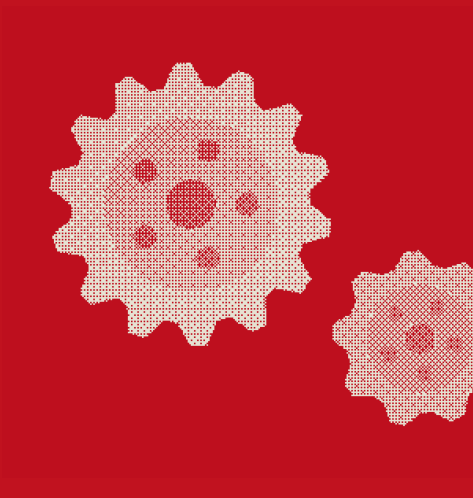


BUILD IT, DON'T RENT IT

When to replace a monthly tool with your own, how to build it with AI, and the checks that make it safe to rely on.

43 PAGES 1 SOURCES 7 TASKS

05



BEFORE YOU START

AI changed the maths of building your own tools.

For years, the rule for a small business was simple: never build software, always rent it. Building meant hiring a developer, and a developer cost more in a month than a subscription cost in years.

AI coding assistants have changed part of that. A non-developer can now describe a small tool in plain language and get working code in an afternoon: a form that writes to a sheet, a script that emails a weekly report, a page that checks a payment before a download.

That doesn't mean you should build everything. Most of what you rent is worth renting. This book is about telling the difference, and then building the few things worth building **safely**: with a spending cap, tests that can actually fail, and a check on the number your plan depends on.

Same format as the other books: a story, the numbers with sources,¹ the build, a case file from a real studio, and one task for today.



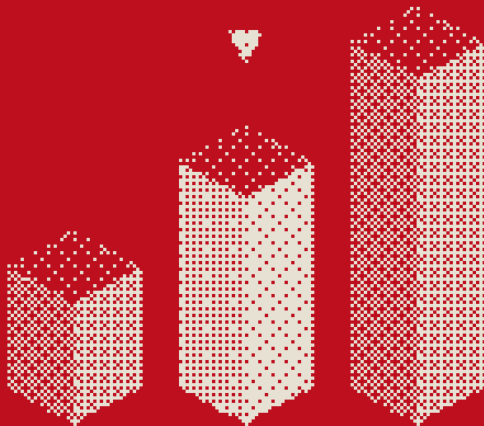
CONTENTS

[01]	The subscription stack What you really pay each month	P. 04
[02]	Rent, build or skip Three questions before you build anything	P. 10
[03]	Tear it down first Learn the tool before you copy it	P. 15
[04]	Build it with AI One script, one job, the cheapest model that works	P. 20
[05]	Tests that can fail Prove your checks catch real mistakes	P. 25
[06]	Guards before go-live Spending caps, dry runs and honest logs	P. 30
[07]	Check the number that matters The ten-minute habit	P. 35
[++]	Sources Every number in this book, traced	P. 41

[01] CHAPTER

THE SUBSCRIPTION STACK

Every tool is cheap on its own. Together, they're often
your biggest cost after your own time.



[01] THE NUMBERS

People guess \$86. The real number was \$219.

DANIEL

In 2022, the market research firm C+R Research asked 1,000 US consumers a quick question: roughly how much do you spend on subscriptions each month? They had ten seconds to answer.¹

The average guess was **\$86**. Then the same people listed every subscription one by one. The real average was **\$219**, about \$133 a month more than they thought. More than half underestimated by at least \$100.

Why? The same survey found that 72% had every subscription on automatic payment, and 74% said recurring charges were easy to forget.

That study was about people's personal spending, not businesses. But the mechanism is the same in a small business, and often stronger, because business tools are added one at a time, each for a good reason, each on autopay. Nobody adds them up.

Daniel did it once. He expected about \$150 a month. The list came to eleven tools and a good deal more than that, including two he'd stopped using months earlier and one that did the same job as another.

So the first step in this book is knowing what you pay.

THE NUMBER

\$133

a month more than people thought they spent on subscriptions.

They guessed \$86. Itemised, it was \$219. 74% said recurring charges were easy to forget.

SOURCE [1]: C+R RESEARCH, SUBSCRIPTION SPENDING SURVEY OF 1,000 US CONSUMERS, 2022.

[01] THE BUILD

The stack sheet: one row per tool.

Pull last month's card and bank statements. Every recurring charge gets a row:

COLUMN	WHAT TO WRITE
Tool and price	The monthly cost, after converting any yearly plan
Job	The one job it does for you, in five words
Used in the last 30 days?	Yes or no. Be honest
Overlaps with	Any other tool that does the same job
Pain if it vanished	None, annoying, or the business stops

Three kinds of row usually appear. **Unused tools**, which you cancel today. **Overlaps**, where you keep the better one. And **simple jobs on expensive tools**, where you're paying for a large product to do one small thing. That third kind is what the rest of this book is about.

DO THIS TODAY

LIST THE STACK

~40 MIN

FREE

BEGINNER

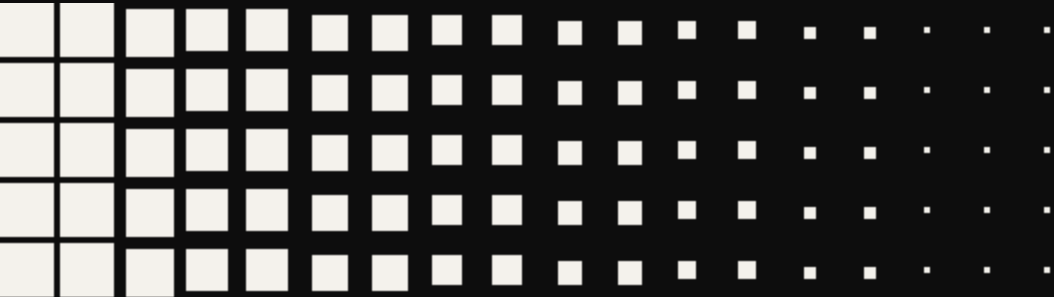
TASK_01.TXT

READY

Find out exactly what you pay for, and what each tool does.

- [] Pull last month's card and bank statements.
- [] List every recurring charge with its monthly cost.
- [] Fill in: job, used in 30 days, overlaps, pain if it vanished.
- [] Cancel anything unused. Keep one of each overlap.
- [] Circle the tools doing one small job for a large price.

You're done when every tool has a row and you've cancelled at least the unused ones.



RECAP

CHAPTER 01 IN THREE LINES

01 People underestimate recurring costs. Autopay hides them.

02 One row per tool: job, use, overlap, pain if it vanished.

03 Cancel the unused. Circle the small jobs on big tools.

NEXT

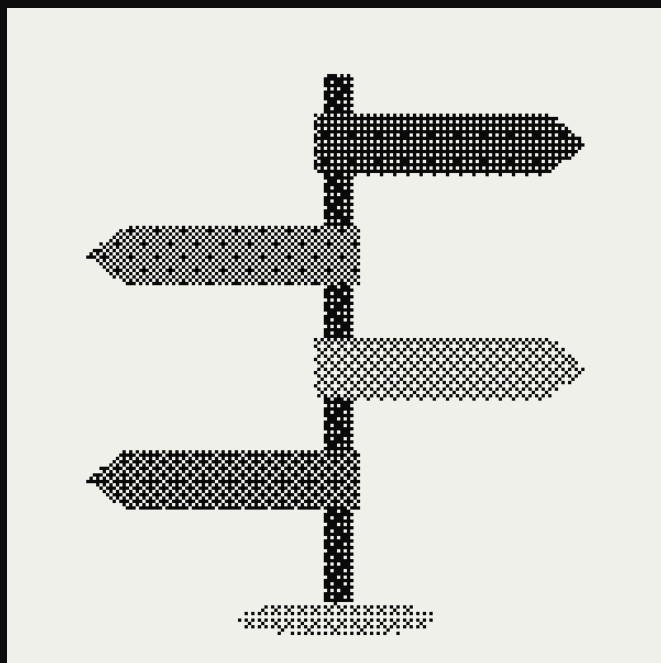
You've found the candidates. Next: deciding which ones are worth building yourself.



[02] CHAPTER

RENT, BUILD OR SKIP

Most tools are worth renting. A few are worth building.
Some jobs aren't worth doing at all.



[02] THE TEST

Three questions before you build.

QUESTION	IF THE ANSWER IS...
Is the job small and well defined?	No: rent it. Big, fuzzy jobs grow into big projects
Is the job core to how you make money?	Yes, and it touches payments, security or customer data: rent a proven tool
Would you be paying mostly for features you never use?	Yes: build the small part you need

And one question before all three: **does the job need doing at all?** The cheapest tool is the one you don't need. A report nobody reads, a sequence nobody replies to, or an integration that saves two minutes a month isn't worth renting or building.

[02] WORKED EXAMPLES

Marcus builds one. Daniel keeps one.

MARCUS

DANIEL

Marcus's stock alert. He paid for an inventory app with forecasting, purchase orders, supplier portals and reports. He used one feature: an email when a product drops below five in stock.

That job is small and well defined. It isn't core in the risky sense: a late alert is an annoyance, not a lost payment. And he was paying almost entirely for features he never opened. So it passed all three questions. A short script that reads his store's stock levels once a day and emails him when anything is low replaced it.

Daniel's course platform. He was tempted to replace it too. It hosts his lessons, handles logins, takes payments, and manages his two-part payment plan.

That job isn't small. It handles money and personal data. And he uses most of what it does. All three answers said *rent*. So he kept it, and spent the time he'd have spent rebuilding it on his content instead.

The point of the test is to build only where building clearly wins, and to stop wondering about the rest.

DO THIS TODAY

SCORE THREE TOOLS

~30 MIN

FREE

BEGINNER

TASK_02.TXTREADY

Decide rent, build or skip for your three most expensive tools.

[]

 Take the three priciest tools from your stack sheet.

[]

 Ask first: does this job need doing at all?

[]

 Then the three questions: small, core, unused features.

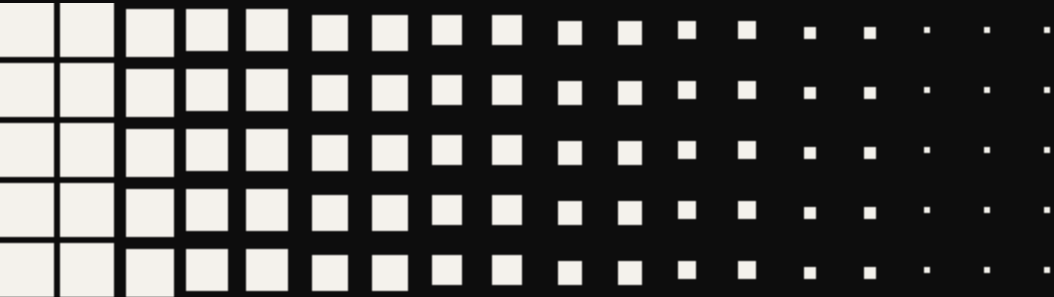
[]

 Write rent, build or skip next to each, with one sentence why.

[]

 Pick at most one 'build' to try this month.

You're done when three tools are scored and you've picked one, or none, to build.



RECAP

CHAPTER 02 IN THREE LINES

- 01 First ask whether the job needs doing at all.
- 02 Build only small, well-defined, non-risky jobs.
- 03 Rent anything that handles money, security or customer data.

NEXT

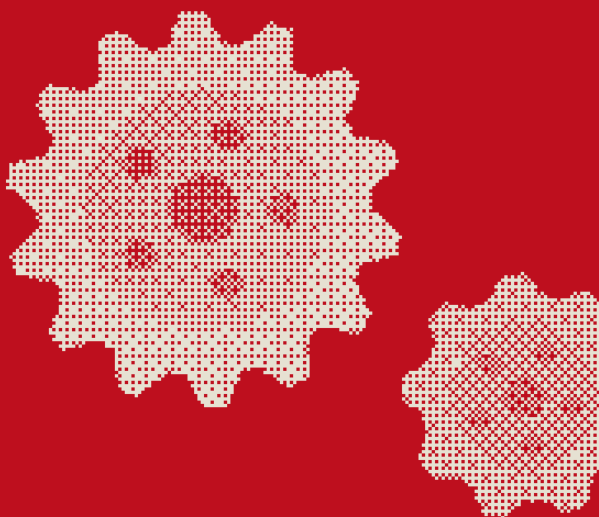
You've picked something to build. Next: learning exactly how the paid version works first.



[03] CHAPTER

TEAR IT DOWN FIRST

Before you rebuild a tool, write down everything it actually does. You'll build less, and build it right.



[03] THE METHOD

A teardown is a list of what the tool really does.

MARCUS

A teardown means going through a tool you pay for and writing down everything about it: every screen, every setting, every price, and the one or two things you actually use it for.

It sounds slow. It saves time, for three reasons:

- + **You find the hidden jobs.** Marcus's inventory app also merged stock counts across two sales channels. He'd never noticed, because it just worked. A rebuild without it would have shown the wrong numbers.
- + **You find the small print in the pricing.** Credits, per-action charges, limits that reset monthly. These tell you which parts are expensive to run, and therefore which parts to build carefully.
- + **You get a spec for free.** Your teardown notes become the instructions you give an AI coding assistant in Chapter 4.

A good teardown has three files:

- [01] **What it does:** every screen and feature, and which ones you use.
- [02] **What it costs:** every price, credit and limit, and what triggers a charge.
- [03] **The rebuild spec:** only the parts you need, written as a list of jobs.

CASE FILE / A REAL BUILD

A paid tool, torn down and rebuilt free.

CASE FILE

A one-person studio picked a popular, paid lead-data tool and rebuilt the parts small businesses use most as a free, open tool anyone can run.

How the teardown was done:

- + **Pass one, free:** every page, menu and setting in the paid tool, written up as a teardown document, a pricing and credits document, and a rebuild spec.
- + **Pass two, small and paid:** a tiny real workflow with a spending cap agreed in advance. The credit balance was recorded before and after, along with each step's cost label, because the tool's usage page gave no per-step breakdown.
- + **A note on surprises:** the tool runs paid enrichments the moment a column is saved, unless that's switched off. The teardown recorded it, so the rebuild could show the cost of a run *before* it starts.

Then the rebuild, guided by the spec, and published as a public code repository with setup instructions, a short build prompt and a full specification.

What made it work was writing everything down before writing any code. The teardown answered most of the design questions in advance, and every lesson from it went into a written playbook that the next teardown starts from.

DO THIS TODAY

TEAR DOWN ONE TOOL

~60 MIN

FREE

BEGINNER

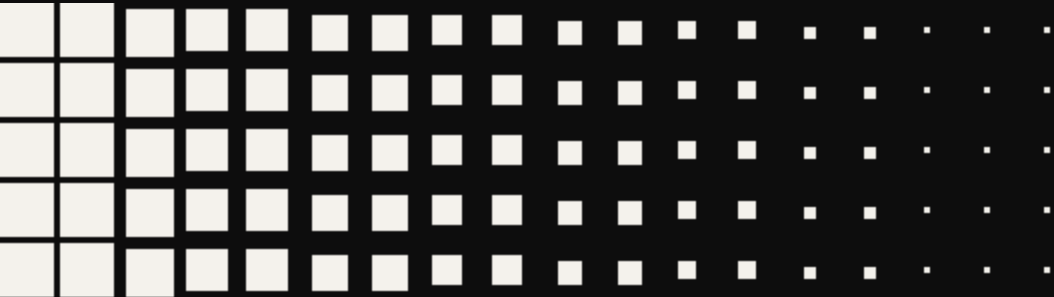
TASK_03.TXT

READY

Write a one-page teardown of the tool you picked in Chapter 2.

- [] Open every screen and settings page. List each feature.
- [] Mark the ones you actually use.
- [] List every price, credit and limit, and what triggers a charge.
- [] Look for jobs it does that you'd never noticed.
- [] Write the rebuild spec: only the jobs you need, one per line.

You're done when you have a one-page spec someone else could build from.



RECAP

CHAPTER 03 IN THREE LINES

01 Write down everything the tool does before you copy it.

02 Find the hidden jobs and the pricing small print.

03 The teardown becomes the spec for the build.

NEXT

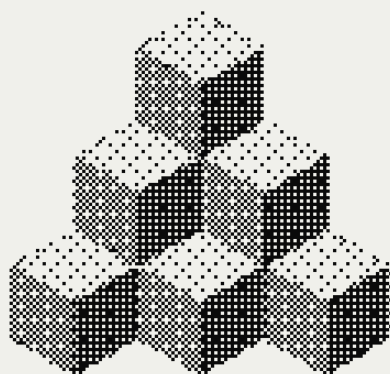
You have a spec. Next: building it with AI, one small job at a time.



[04] CHAPTER

BUILD IT WITH AI

One script, one job, and the cheapest model that does it well.



[04] THE APPROACH

Small pieces, each one testable.

MARCUS

AI coding assistants such as Claude Code, Cursor and others can write working software from a plain description. They work best when the job is small and the description is precise. So the method is:

- 1. One script, one job.** "Read stock levels from my store and email me when any product is under five." Not "build me an inventory system".
- 2. Talk to services directly.** Most tools you use, from your store to your email to your payment provider, offer an official API: a documented way for code to read and write data. A short script calling an API is easier to understand and cheaper to run than a chain of connected apps.
- 3. Keep keys out of the code.** API keys go in a separate settings file that's never shared or uploaded. Your assistant should read them from there, and never print them.
- 4. Ask for a test with every piece.** "Also write a test that proves the email is sent when stock is four, and not when it's six." Chapter 5 explains why.
- 5. Run it yourself before you trust it.** On a copy of your data or a test mode first, then for real while you watch.

Marcus's stock alert took one afternoon with an AI assistant. It runs once a day on a free scheduler.

CASE FILE / A REAL BUILD

The cheapest model that does the job.

CASE FILE

A one-person studio runs a lot of AI tasks: sorting, summarising, drafting, classifying. Paying top prices for every one would add up fast. So it routes each task to the cheapest model that does it well, through one small shared helper.

How it works:

- + **Tedious, simple tasks** such as classifying a message or drafting a short reply go to a provider's **free tier**, with a fast open model.
- + **Hard tasks** such as writing, reasoning or reviewing go to a top model.
- + **One helper file** handles both, so every script in the studio asks for "cheap" or "best", not a specific model.
- + **Limits are written in:** the free tier allows about 1,000 output tokens a minute on its default model, so the helper keeps responses short and waits politely when told to slow down.
- + **Prompts are tested before they're trusted:** a prompt is run ten times and scored, and a change is kept only if the pass rate goes up.

What made it work was making the cheap path the default. Most tasks never need the best model, and the studio only pays for one when a task has earned it.

DO THIS TODAY

BUILD ONE TINY SCRIPT

~60 MIN FREE BEGINNER

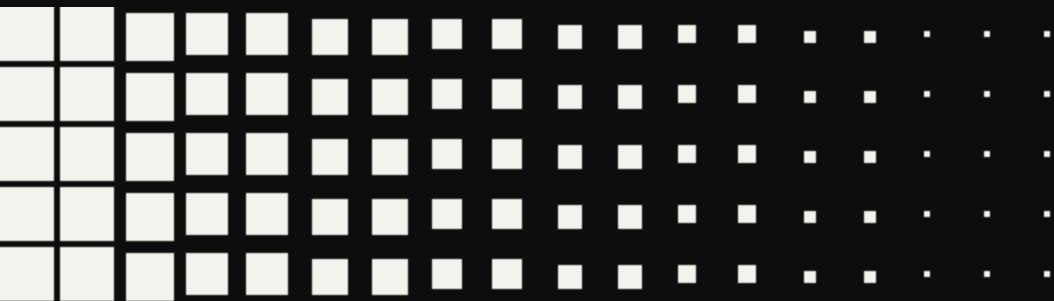
TASK_04.TXT

READY

Build the smallest useful version of the job from your spec.

- [] Pick one job from your spec, the smallest that's still useful.
- [] Describe it to an AI coding assistant in two or three sentences.
- [] Put any API key in a settings file, never in the code or the chat.
- [] Ask for one test that proves it works, and one that proves it says no.
- [] Run it in test mode while you watch.

You're done when the script runs once, correctly, and its test passes.



RECAP

CHAPTER 04 IN THREE LINES

01 One script, one job, a precise description.

02 Use official APIs, and keep keys out of the code.

03 Route each task to the cheapest model that does it well.

NEXT

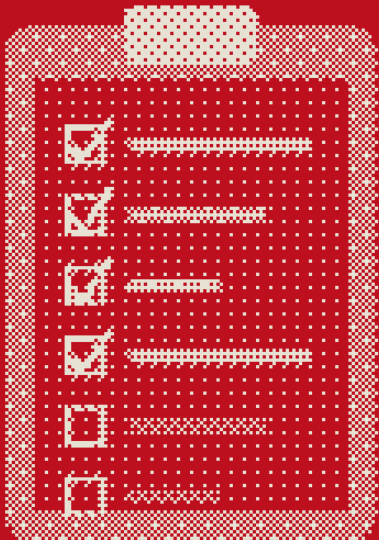
Your script works. Next: proving its checks would catch a real mistake.



[05] CHAPTER

TESTS THAT CAN FAIL

A test that always passes proves nothing. Break your code on purpose and make sure the test notices.



[05] THE IDEA

A green tick can lie.

MARCUS

When an AI assistant writes code, it usually writes tests too, and they usually pass. That's reassuring, and sometimes it means nothing.

A test can pass because the code is right. It can also pass because the test checks the wrong thing, or checks nothing at all. For a tool that handles your money or your customers, you need to know which.

There's a simple way to find out, and it's used by professional engineers: **break the code on purpose and see whether a test fails**. It's called mutation testing.

Take Marcus's stock alert. The rule is "email me when stock is under five". Change the five to fifty and run the tests. If they all still pass, no test is really checking the threshold. Change "under" to "over". Remove the email step. Each planted mistake should make at least one test fail.

Three rules make this work:

- + **Give each rule an input only it can catch.** If two checks could both catch the same example, you can't tell which one is working.
- + **Plant mistakes in behaviour, not spelling.** Code that won't run proves nothing.
- + **Add a planted mistake for every safety check you write.**

CASE FILE / A REAL BUILD

Side by side, before launch.

CASE FILE

When a one-person studio rebuilt a paid tool as a free one, the rebuild had over 200 automated tests, all passing. Before publishing, it ran one more check.

The side-by-side run:

- + **The same three real records** went through the original paid tool and the rebuild.
- + **The same inputs, the same steps**, one after the other.
- + **Every difference was written down** in a comparison document.

What it found: ten real bugs that the 200 passing tests had not caught, all fixed before anyone else used the tool.

The studio also runs a mutation harness on that codebase: a script that plants deliberate mistakes in the code and requires every one of them to be caught by a test. A planted mistake that survives means a missing test, and it gets written before the work is called done.

What made it work was treating "all tests pass" as the start of checking, not the end. Comparing against the real thing, with real data, finds what tests written by the same mind can miss. The side-by-side run is now a required step in the studio's playbook, done *before* publishing.

DO THIS TODAY

BREAK IT ON PURPOSE

~40 MIN

FREE

BEGINNER

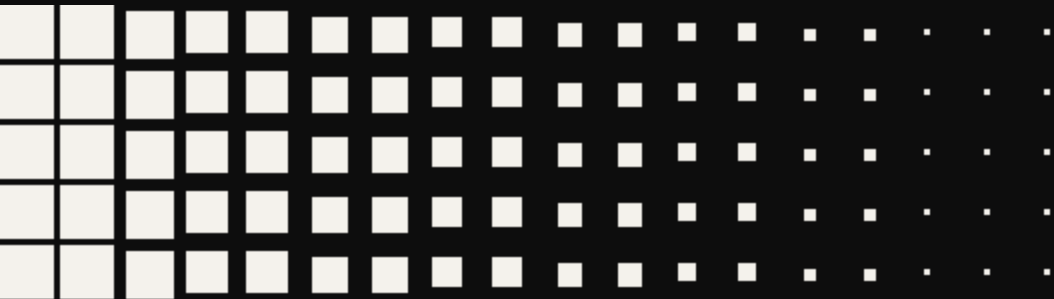
TASK_05.TXT

READY

Prove your script's tests would catch a real mistake.

- [] Run your tests. Note that they pass.
- [] Change one rule in the code: a number, a comparison, a step.
- [] Run the tests again. At least one must fail.
- [] If none fail, ask your assistant to write the missing test.
- [] Undo the change. Repeat for every rule that matters.

You're done when every planted mistake makes at least one test fail.



RECAP

CHAPTER 05 IN THREE LINES

01 Passing tests can mean nothing. Prove they can fail.

02 Plant a mistake for every rule, and require a failure.

03 Compare against the real tool with real data before launch.

NEXT

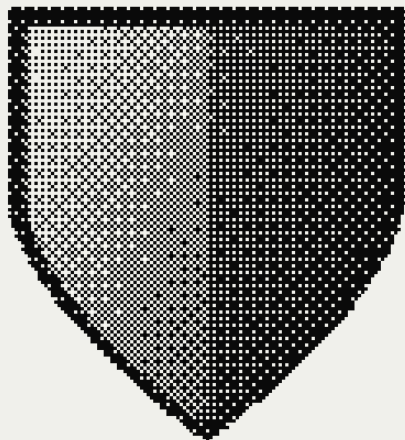
Your tests are honest. Next: the guards that stop a script from costing you while you sleep.



[06] CHAPTER

GUARDS BEFORE GO-LIVE

A script that spends money runs at machine speed. Put the limits in before it runs, not after.



[06] THE BUILD

Four guards for anything that spends.

DANIEL

Any script that calls a paid service, sends a message or charges a card can do in a minute what you'd never do by hand in a month. So it needs limits that don't depend on you watching.

1. A spending cap per run. Before the script starts, it works out the most the run could cost, and refuses to start if that's over your limit. Show that number before every run.

2. A dry run first. A mode that does everything except the paid or outgoing step, and prints what it *would* have done. Every new script runs dry before it runs real.

3. A go before any paid run. For anything metered, a person says "go" after seeing the dry run and the maximum cost. Describing a plan is not the same as approving it.

4. An honest log. After every run: what was done, what it really cost according to the provider, and what failed. An error is logged as an error, never as "no result".

Daniel's version is small. His script that sends the Monday report has a dry-run switch, can send at most one email per run, and writes a line to a log file each Monday. That's enough for a script that small. A script that spends money needs all four.

CASE FILE / A REAL BUILD

Money in micro-dollars.

CASE FILE

A one-person studio builds tools that call paid data services. Its rules for anything that spends money are written into a playbook and enforced in code.

The rules:

- + **Money is stored as whole numbers of micro-dollars**, never as decimals, so repeated small charges never drift through rounding.
- + **Every paid call has a maximum cost** sent with the request, so the provider refuses anything over it.
- + **The real charge is recorded after each call**, from the provider's own answer, not from a price list.
- + **A run reserves its worst-case cost before it starts**, so two runs at once can't both spend the same budget.
- + **The interface shows the most a run can cost** before the person presses the button.
- + **An error is never treated as an answer**. A service that's down returns "try again", not "no result", so nothing paid for is thrown away.

What made it work was deciding the limits before writing the feature. Each rule is small. Together, they mean a run can't cost more than the person agreed to, and every cent spent is accounted for.

DO THIS TODAY

ADD A CAP

~40 MIN

FREE

BEGINNER

TASK_06.TXTREADY

Make sure one automation can't spend or send more than you allow.

[]

 Pick one script or automation that spends or sends.

[]

 Write its maximum per run: messages, credits or dollars.

[]

 Add a check that stops the run if it would go over.

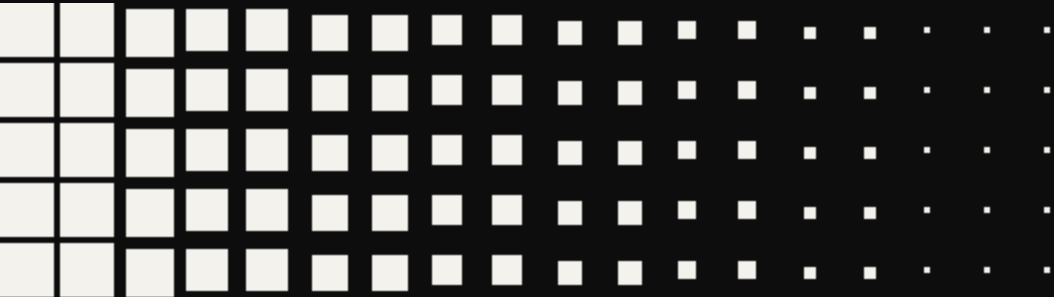
[]

 Add a dry-run mode that prints what it would do.

[]

 Add one line to a log after every real run.

You're done when a dry run shows the maximum cost, and a test run stops at the cap.



RECAP

CHAPTER 06 IN THREE LINES

01 Scripts spend at machine speed. Cap them in advance.

02 Dry run first, then a person's go, then the real run.

03 Log the real cost, and never treat an error as an answer.

NEXT

Last chapter: the one check that matters more than any code.



[07] CHAPTER

CHECK THE NUMBER THAT MATTERS

Every plan rests on one figure. Ten minutes checking it
can save months building on it.



[07] THE IDEA

One number holds the plan up.

Every business plan, every tool you build, and every offer you sell rests on a number. "Shop owners will pay \$X for this." "This saves five hours a week." "Most customers in this niche have problem Y." If that number is wrong, everything built on it is wrong too, however well it's built.

These numbers often come from somewhere convenient: a vendor's blog, a viral post, a statistic everyone repeats. Vendors have a reason to make their problem sound big. Viral posts have a reason to sound dramatic. Neither is a reason to trust the figure.

So before building anything that depends on a number, do three things:

- [01] **Find the original source.** Not the article that quotes it. The study, the survey, the official data.
- [02] **Check who measured it, and how.** A vendor surveying its own customers measures something different from a government survey of every business.
- [03] **Check it fits your case.** A figure about large companies may not describe a two-person studio. A US figure may not describe your market.

It usually takes about ten minutes. Most of the time, the number holds up. When it doesn't, you've just saved yourself months.

CASE FILE / A REAL BUILD

The ten-minute check, as a rule.

CASE FILE

A one-person studio made checking the load-bearing number a standing rule, after the habit proved its worth.

The rule:

- + **Every number carries its source**, written next to it, wherever it appears: a plan, a post, a book, a price.
- + **A vendor's figure is labelled as a vendor's figure** in the same sentence.
- + **Before building on a number, check it.** If it can't be traced, it goes in a notes file marked "unverified", never into a plan.

The check that earned the rule: the studio was preparing an offer for a type of business service. The opportunity rested on one percentage published by companies that sold that service. A ten-minute check of where the figure came from, and what it really measured, showed the real opportunity per client was a small fraction of what the figure implied.

The studio didn't build the offer. It kept the engine it had already built for something better, and wrote the lesson down.

What made it work was doing the check *before* the building, when changing course cost ten minutes instead of months. Every book in this library follows the same rule: every number is traced on the sources page.

DO THIS TODAY

CHECK YOUR NUMBER

~20 MIN

FREE

BEGINNER

TASK_07.TXTREADY

Find the one number your current plan depends on, and check it.

[]

Write down your current plan in two sentences.

[]

Underline the number it depends on most.

[]

Find where that number really came from.

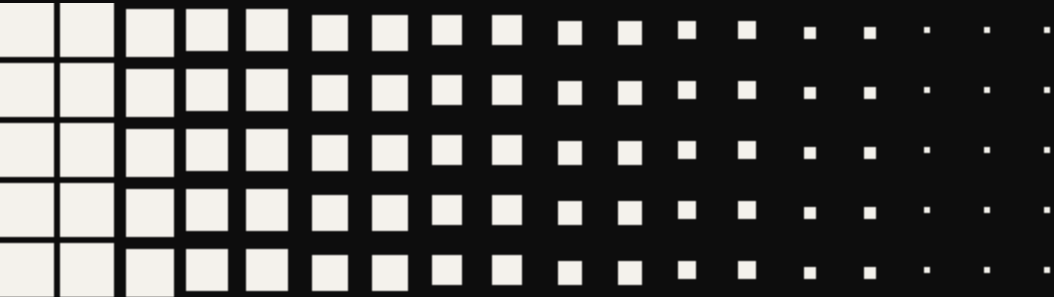
[]

Ask: who measured it, how, and does it fit my case?

[]

Write the source next to it, or mark it unverified.

You're done when your key number has a real source next to it, or you've changed the plan.



RECAP

CHAPTER 07 IN THREE LINES

- 01

Every plan rests on one number. Find it.
- 02

Trace it to the original source, and check it fits.
- 03

Ten minutes before building beats months after.

NEXT

That's building your own tools. Book 6 covers the sales call and the proposal after it.



[++] WHERE YOU ARE

NOW OWN IT.

If you did the seven tasks, you now have:

- [01] A sheet of every tool you pay for.
- [02] Rent, build or skip, decided for your biggest three.
- [03] A teardown and a spec for one tool.
- [04] One small script that does one job.
- [05] Tests you've proved can fail.
- [06] A cap, a dry run and a log on one automation.
- [07] The number your plan depends on, checked.

Across these five books, that's an inbox that answers itself, a content engine, a lead pipeline, a back office that runs at night, and the skills to build and check your own tools. Book 6 covers the step that brings the work in: turning a sales call into a signed proposal.

None of it needs to happen at once. The 9-week plan in this library takes one system a week, in the order that frees the most time first. Start with week one.

SOURCES

Every number, traced.

-
- [1] C+R Research, subscription spending survey of 1,000 US consumers, conducted late April to early May 2022: estimated \$86 a month, itemised \$219; 54% underestimated by \$100 or more; 72% on auto-pay; 74% said recurring charges were easy to forget. ciresearch.com/blog/true-cost-of-subscriptions; reported by CNBC, Sep 6, 2022.
-
- [--] Tools named in this book were current in October 2026. Features and prices change: check each one before you pay.
-
- [--] Case files describe real practice from a one-person studio, including its written teardown playbook. They are shown as examples of method, not as promised results.
-
- [--] Daniel and Marcus are composite characters. Nothing in this book is a promise of money saved or earned.

[++] THE OPERATOR'S LIBRARY

This book is one of six.

Here's the whole library.

You've read Book 5 for free, and it stays free. It's part of **The Operator's Library**: six field manuals and a 9-week plan for running a small online business on AI systems, in the same format as this book.

PART	WHAT IT COVERS
Book 1	The Inbox That Answers Itself. Email, DMs and customer questions answered fast, with a person in charge
Book 2	The Content Engine. One idea a week, picked from demand, turned into a week of posts
Book 3	Leads on Autopilot. Find the right people, check they're real, follow up with every one
Book 4	The Back Office That Runs at Night. Booking, reminders, onboarding, invoices and delivery
Book 5 (this one)	Build It, Don't Rent It. When to replace a monthly tool with your own, and how to make it safe
Book 6	From Call to Contract. Sales calls and proposals, from prep to a same-day proposal
Bonus	The 9-Week Automation Plan. All six books in order, one system a week, with a weekly scorecard

The whole library is **\$200, one payment**, as PDF downloads. It includes the two free books, so you only need it if you want the other four and the plan.

Everything is at **books.clientoptimal.com**, with sample pages. If this book was enough for now, that's fine too. Start with the first task and keep going.

THE OPERATOR'S LIBRARY / NEXT

BOOK 06 FROM CALL TO CONTRACT

Your systems run the business. Book 6 covers the part that brings the work in: preparing for sales calls, asking better questions, and sending the proposal the same day.

