TSPS · EL PASO · 2026 · SESSION HANDOUT

Build Your Own Tools

Practical AI-powered app-building for land surveyors — the depth behind the talk, in the order the talk ran.

John Andricopoulos, RPLS, PMOCP, CAIKMP
Founder, Apollo Lens Solutions · apollolens.io · info@apollolens.io

HOW TO USE THIS

The slides carried headlines. This carries the detail: every aside I skipped for time, the prompts you can paste as-is, the checks to run before anyone else uses a tool, and a worksheet to find your first project. Pages follow the six-step build cycle, so the dots in each page header tell you which step you're reading about.

Nothing in here requires a programming background. It requires the thing you already do for a living: describing a job so precisely that a stranger can retrace it.

THE ONE IDEA

"You've been writing precise plain-language specifications your whole career — a metes-and-bounds description is one. AI turned that skill into the ability to build software."

WHAT'S INSIDE

Six steps, in a loop — and you already run every one of them on a survey job

Setup → Plan → Execute → Review → Deploy → Iterate, and Iterate feeds back into Plan. It's a loop, not a line. This is the whole method; the rest of the handout is the depth under each step.

STEP	WHAT IT MEANS, IN PLAIN ENGLISH	SURVEY-JOB EQUIVALENT	THE RULE
1 Setup	Pick the tool and install it. Write the standing instructions the AI reads every session. Set the firm's data rule. Give the work a home (one folder per tool).	Mobilization — gear, control, the job file	Set your data rule and your instructions file **before anyone builds anything**.
2 Plan	One-sentence problem, scoped to one task, plus your domain knowledge in writing. Run the build-vs-buy check first.	Research + control	One problem. One task. Your knowledge, written down.
3 Execute	Let the AI draft — one feature at a time, or the whole tool in one pass, then revise section by section. Both work. Revising is not optional.	Fieldwork + computation	Draft, then revise. One focused change per request.
4 Review	Verify against known answers. Tests the AI can't talk its way past. Then a second check: is it *safe*, not just working?	QC — independent of the person who did the work	Known answers and independent checks. Confident wording is not evidence.
5 Deploy	Get it into the hands of the people who'll use it, in a form that doesn't need IT. Simplest: one HTML file. Include the ten-minute walkthrough.	Delivery	One file, no installs, walk the staff through it.
6 Iterate	Fix what breaks, add what's missing, keep it maintainable. This is where most tools quietly die.	Revisions — the file you can pick up in five years	Keep the "why" written down so anyone can pick it up.

> **The half of the loop that matters most, in one breath:** one problem, one task, your knowledge — and verify it like a boundary.

How these tools behave — three habits worth writing on a sticky note

1

A full context loses accuracy

The more that's on the page, the more gets overlooked — like the last page of a long day's field book. The fix is cheap: start a fresh conversation for the next task, and let the standing-instructions file carry the memory.

2

One task at a time

Sharp on one focused ask; sloppy on five bundled together. "Add this feature" — not "add this, and also reorganize, and also change the colors." Scope creep ruins AI builds exactly the way it ruins projects.

3

Structure beats a brain dump

The same instructions land better as headings and numbered rules than as a paragraph. That's why every instruction file is markdown, and why the Plan step looks a lot like writing a scope of services.

What changed, and what it doesn't change

Chatbots answer questions. Agents do work. An agent writes the files, runs the code, tests it, reads the error, and fixes its own mistake while you watch and steer. It can touch your files — that's the difference between chat in a browser tab and Claude Code or Codex running on your machine.

You've spent your career waiting on the vendor. You don't have to anymore. If it wasn't in the release notes, you waited — a year, five years, forever. Now you can solve your own headaches, on your own or with a little help. The limit isn't a programming background or budget. It's imagination — knowing what's worth building.

The goal is efficiency, not job elimination. These tools remove the tedious, repetitive, nobody-likes-doing-this work around your judgment — not the people, and never the judgment (page 12).

The AI writes different code every day. The tool gives the right answer every time. There are several valid ways to code a correct answer, so it doesn't matter that Tuesday's code differs from Monday's. What matters is that the finished tool is deterministic and testable — and once it's built, you can take the AI out of it entirely. Both demo tools have no AI inside them.

You do the surveying-shaped work. The AI does the typing-shaped work. You define the problem, scope it, supply the domain knowledge, and check the result. The AI writes, runs, tests, and fixes. Neither succeeds alone: a surveyor with no AI is slow; an AI with no surveyor builds the wrong thing beautifully.

Five words cover most of what you'll hear — with the survey equivalent

Five of these opened the talk; agent and hallucination were defined when they came up. Each gets a plain-English line and, in italics, the survey anchor that makes it stick. The rest of the glossary is on the next page.

Markdown `.md`	A plain text file with light formatting: a `#` makes a heading, a `-` makes a bullet. Opens in Notepad. Every instruction file you'll see is one of these. *Field notes, typed.*
Standing instructions `CLAUDE.md · AGENTS.md`	A markdown file the AI reads before every session: your firm, your units, your file rules, what's off-limits, how you want things explained. You write it once; every session inherits it. (Sample on page 5.) *The desk binder for a new hire.*
Skill `SKILL.md`	A smaller instruction file for one repeatable task — "here's how we format a point file" — that the AI loads only when that task comes up. **Awareness flag:** a skill is instructions the AI will follow and, often, code it will run, with every permission the AI has. Treat a downloaded one like a downloaded program (page 9). *An SOP sheet.*
Token	The unit these tools read, write, and bill in — roughly three-quarters of a word. The price tier you buy is a token budget; a conversation's memory is measured in them too. You never count them; you just need to know that's what the meter runs on. *The "foot" of AI — cost and memory are counted in it.*
Context window	Everything the AI can see at once in one conversation: what you pasted in, the files you showed it, the standing instructions, and what it has said back. Finite. It does not know what's on your hard drive or what you told it last Tuesday. *One field book per job.*
Agent	An AI that does work rather than just answering — writes files, runs code, tests it, fixes its own errors — on your machine, with your files, while you steer. Claude Code, Cowork, and Codex are agents; the chat window is not. *Handing a tech the job, not asking how they'd do it.*
Hallucination	Output the AI made up and worded as if it were settled fact. These tools are astonishingly good at it. Confident wording is not evidence — which is why Review is the longest block in the method. *A deed that says "more or less."*

The rest of the words — defined where they came up in the talk

Dependency came up in the Plan block; the rest came up inside the Review, Deploy, and Iterate blocks. Same format: plain English first, survey anchor in italics.

Prompt	What you type to the AI. A good one reads like a scope of services: problem, input, output, rules, and how you'll know it's done. (Template on page 6; real examples on page 7.) *The work order.*
Dependency	A chunk of someone else's software your tool leans on — a library that reads DXF files, a map component, a PDF parser. It changes, breaks, or vanishes on its schedule, not yours. Every piece of borrowed code is a problem you now own. *Control you didn't set and can't re-observe.*
Single-file tool `one .html file`	A tool that is one HTML file: runs from a folder in any browser, offline, no installs, no server. It lives in the browser's sandbox, so it can only work on what the user hands it. Almost no dependencies. Start here. *A hand-held calculator, not a server room.*
Local server `localhost`	A small program running on your own computer that a browser page talks to. It's what the AI usually builds when you ask for "an app." It's also a door: anything that can send it a request can drive it (page 9). *A gate in the fence — useful, and now something to lock.*
Test `regression test`	A small piece of code that runs the tool against a known input and confirms the output — every time anything changes. A regression test is one you keep forever so an old bug can't sneak back in. Tests check the math, not the look. *A check that reruns itself. A control check, automated.*
Version control `Git · GitHub · commit`	A record of every change to the tool, each one reversible. A *commit* is one saved snapshot with a note; GitHub is where the record lives. Commit before every AI edit — the undo button is what makes experimenting safe. *The tool's own field book: someone else can retrace how it got here.*
Database	Shared memory. A spreadsheet that many tools and many people can read and write at once without stepping on each other. You need one the moment two people need the same data, or the tool must remember yesterday. *The firm's project index, not a field book in one truck.*
Authentication `sign-in`	Who's allowed in, and what each person may see or change. Matters the instant a tool leaves one laptop — especially with client data. *Who has keys to the plan room.*
Decisions file `DECISIONS.md`	A running list of "we chose X because Y." Stops the next person — or the next AI session — from undoing a choice you made on purpose. *The note in the file that explains why you held the found monument.*
Model vs. product	The model is the AI engine (version names churn monthly); the product is what you buy and use — Claude, Claude Code, Cowork, ChatGPT, Codex. Talk about products. Nothing in this handout depends on which model version is current — but within a product you still pick a model for each session: the reasoning model for planning and diagnosis, the fast one for small edits (habit 1, page 7). *The total station model number changes; "the instrument" doesn't.*
Prompt injection	Text hidden inside something the AI reads — a web page, a file, a skill — that tries to give it instructions. The reason "what does it trust?" is a Review question (page 9), and the reason skills are read before they're installed. *A forged note in the job file telling the crew to move a monument.*

Setup happens once. Every tool after that inherits it.

People skip this because it isn't the fun part. Don't. Setup is an hour, once — and every tool you build afterward starts from a running start instead of from zero. Four items; two of them are load-bearing.

1. **Pick the tool. Install it.**
   Claude or ChatGPT — $20 a month either way (page 14). Install the desktop app. When a bigger build eventually needs software you don't have (Python, say), let the AI walk you through the install. Just stay awake about what permissions you grant along the way: **"the AI asked" is not a reason to say yes.**

2. **Write the standing instructions.**
   The single highest-leverage item on this page. A plain-text file the AI reads at the start of every session — `CLAUDE.md`, `AGENTS.md`, whatever your tool expects. Your firm, your units, your coordinate system, your file rules, what's off-limits, how you want things explained. Honest warning: **this is not intuitive at first.** Force yourself through the learning curve; it's a game changer for result quality.

3. **Decide what the AI can see — before anyone pastes a deed in.**
   When you paste something into one of these tools, it goes to the vendor's servers. On the individual plans (the $20 tier) the vendor may use your content to train future models unless you opt out in settings; on the business plans they don't by default. So: three decisions, made once, by the owner, not the builder — **which plan** the firm is on · **whether the opt-out** is set · **what client material** staff may put in at all, versus what gets built and tested on synthetic or your-own data. Ten minutes. It has to happen first.

4. **Give the work a home.**
   One folder per tool. Not the shared drive, not your desktop, not inside a job folder. Later, when a tool matters, that folder gets version history (page 10).

Data-training terms: Claude Pro/Max "Model training: opt-out"; Team/Enterprise "No model training on your content by default" (claude.com/pricing). ChatGPT Free/Go/Plus/Pro: content used for training by default, opt-out available; Business/Enterprise/API not used by default (chatgpt.com/pricing; openai.com/enterprise-privacy). Verified 2026-08-28, re-checked 2026-09-16. Terms change at vendor discretion — read the plan you're on.

A REAL STANDING-INSTRUCTIONS FILE — STARTER VERSION

```
# About our firm
- Units: US survey feet. Grid: TX North Central
(4202) unless told otherwise.
- Point codes: see codes.csv. Unknown codes get
flagged, never guessed.
- Deliverables go to Civil 3D; match our template
column order (P,E,N,Z,D).

# Rules
- Never modify an original job file. Work on a
copy.
- Never paste credentials. Ask before installing
anything or reaching the internet.
- Build as a single HTML file if the job allows
it; tell me if it doesn't, and why.
- One feature per request. Confirm the plan
before writing code.
- Explain every change in plain English before
the code.

# Data rule
- Synthetic or firm-owned data only in this tool.
No client documents unless the owner has approved
the plan tier and opt-out settings.
```

Copy this, then replace every line with your firm's truth. The AI will help you write it — the first prompt on page 7 is exactly that ask. Add to it every time the AI gets something wrong twice.

WHAT GOES IN IT, OVER TIME

- **Conventions** — units, zones, code lists, file naming, template column orders.
- **Off-limits** — original job files, credentials, the shared drive.
- **How you want it explained** — "layman and technical, every non-trivial block."
- **Hard rules from Review** — "if this test ever fails, the fix is wrong, not the test."
- **Pointers, not paragraphs** — "see codes.csv," "see DECISIONS.md." Keep the file short; put the detail in files it points at.

If a new hire couldn't act on your sentence, the AI can't either

Planning a tool is writing a specification — precise, unambiguous, plain English — and you already do that for a living. Up-front planning is the single most critical success factor, exactly like the projects you run.

NOT READY

"Help me with deed research." A new hire would come back with three questions. So will the AI.

READY TO BUILD

"Given a folder of deed PDFs, pull grantor, grantee, and recording info into one spreadsheet."

Step zero: build or buy?

If software you already own — or something affordable off the shelf — solves 70–80% of the problem, use it. Custom is more expensive than it looks once you count maintenance (page 12), and a vendor product's dependencies are the vendor's problem; a custom tool's are yours.

BUILD WHEN…	BUY (OR KEEP) WHEN…
The workflow is genuinely firm-specific — your codes, your template, your county's rules.	A product you own does most of it and the gap is a habit, not a feature.
The tool doesn't exist, or exists at a price wildly out of proportion to the task.	The task touches licensing, lock-in, or a vendor format you'd have to reverse-engineer.
Owning it outright matters — no subscription, no vendor roadmap, runs offline.	You can't name who will maintain it in year two.
It's one task, one input, one output — a utility, not a platform.	You're asking for a "system." Start with the utility inside it.

Two things to watch the whole way through

Context. *"If it lives in your head or an arbitrary hard-to-find file, the AI probably can't use it."* The AI knows Python; it does not know your point-code library, your county's plat rules, or that you use US survey feet. If the answer lives in Bob's head or in `FINAL_FINAL_use_this_one_v3.dwg`, the AI can't reach it. Getting that knowledge into writing isn't a chore before the build — it *is* part of the build, and it outlives the tool.

Modularity. Small pieces, one job each, clean hand-offs — not one 2,000-line file. You'd never run a long differential level as one loop; you break it into loops that each close on their own, so a bust is findable and contained. Same discipline, same reason.

What it stands on. Decide before you build whether the tool can do its whole job with only what the user hands it — then it's one file, and one file borrows almost nothing. Single-file test and dependencies: page 10. The sentence for the prompt: prompt 5, page 7.

The spec template — fill this in before you type a prompt

PROBLEM one sentence a new hire could act on

INPUT what goes in — and attach a real sample

OUTPUT what comes out, in what format

RULES your domain knowledge: units, codes, never-touch-the-original

DONE WHEN a test against a job you already did by hand

WORKED EXAMPLE — A POINT-FILE REFORMATTER

Problem Convert the data-collector export (P,N,E,Z,D) into the CSV our Civil 3D template imports (P,E,N,Z,D) with descriptions mapped to our office code list.
Input One .csv per job — sample from job 24-118 attached.
Output One .csv: columns reordered, codes mapped. Unknown codes flagged, never guessed.
Rules US survey feet. Point numbers unchanged. Never modify the original file.
Done when The 24-118 sample matches my hand-checked file, zero unmapped codes.

Compare: *"Can you make me something that fixes my point files so they import into Civil 3D right?"* — works on the one file you had open, breaks on the next.

Pro move: **paste your brain dump and ask the AI to reformat it into this structure first — then build from the structured version. (Prompt on page 7.)**

Prompts you can paste as-is — and the six habits behind them

None of these contains code. None needed to. Every one is a sentence a new hire could act on. The first is the real first prompt behind the Point Data Converter, word for word.

1 · SETUP — ASK FOR THE BINDER, NOT THE TOOL

"I would like to use Claude Code to build a single-file HTML application. Please give me some comprehensive instructions I can place in a Claude.md file to help keep things on track. Feel free to ask me any clarifying questions that will help make these instructions more useful."

It doesn't ask for the tool. It asks for the standing instructions and invites questions first. The AI asked three (what kind of app, what technologies, does it talk to a backend); the answers became six hard rules.

2 · PLAN — BRAIN DUMP TO SPEC

"Here is everything I know about the problem, unorganized: [paste]. Reformat this into a clear spec with sections for Problem, Input, Output, Rules, and Done-when. Ask me anything that's ambiguous before you write it."

3 · PLAN — SCOPE CHECK BEFORE YOU COMMIT

"Will this context work if I need to be able to read and upload pdf, csv, and xlsx files and export csv files?"

The honest answer — "not without borrowed code" — forced a decision: CSV in, CSV out. That one question kept the tool a single file.

4 · EXECUTE — PLAN FIRST, THEN BUILD

"Before writing any code: how would you approach this? Tell me the pieces, what each one does, and what could go wrong. Wait for my go-ahead."

5 · EXECUTE — THE SINGLE-FILE ASK

"Build this as a single HTML file that runs from a folder in any browser with no server and no installs, if the job allows it — and tell me if it doesn't, and why."

6 · EXECUTE — WHEN IT'S STUCK

"Stop. Ignore the last three attempts. Here is the requirement restated: [one sentence]. Here is the actual input file and the actual error text. Tell me what's ambiguous before you try again."

7 · ITERATE — PLAIN-ENGLISH RECORD

"Explain what you just changed in plain English, as if to a party chief, and add a line to DECISIONS.md: what we chose and why."

Six habits separate useful tools from expensive toys

1. **Be model-conscious.** The models aren't interchangeable: the deep reasoning model for planning and diagnosis, the fast, cheap one for small edits once you know what you want. The heavy model for everything burns your limits; the light one for the hard thinking is how people decide AI "isn't very good."
2. **One change per request.** "Add this feature." Not "add this, and also reorganize, and also change the colors." Scope creep ruins AI builds the way it ruins projects.
3. **Ask "how would you approach this?" before "build it."** Thirty seconds of the AI explaining its plan is the cheapest place to catch a wrong assumption — same reason you talk through a boundary resolution before drafting the plat.
4. **Give it the real file, not a description.** A sample point file. The actual error message. A screenshot of the plat. Never "it's a CSV with some columns."
5. **When it's stuck, restate — don't argue.** Three failed attempts means your spec is ambiguous, not that the AI is stupid. Rewrite the ask. Don't repeat it louder.
6. **Make it explain itself in plain English.** Demand the layman's explanation alongside the code, every time. You can't QC what you don't understand — or maintain what you can't explain.

\+ **And one more, for the record: save your prompts and decisions like field notes.** The record of *why* lets you — or the AI, next session — pick the work back up (prompt 7; page 12).

TWO WORKING STYLES, BOTH VALID

Feature by feature — build one piece, test it, build the next. Safest for a first tool. **Whole draft, then revise** — have the AI draft the entire tool in one pass, then revise it section by section. Non-negotiable either way: never accept the first draft untested, and make each revision one focused change.

THE BOUNDARY TRACKER KICKOFF, EXCERPTED — SAME DISCIPLINE, BIGGER TOOL

"...produce an implementation plan for Phase 0.5-S only... Once I approve the implementation plan, proceed directly into executing it in this session. Everything you need to know is below and in the referenced documents — the product/requirements phase is complete and locked."

Plan, approve, then execute — and it points at the spec instead of restating it. A scope of work, in other words.

“It looks right” isn't a QC standard

Wrong by 300 miles is safe — everyone sees it in ten seconds. Wrong by 40 feet is expensive — the plat looks right, the closure looks right, and it's found when it costs someone money, with your seal on it. AI-built tools fail the second way, almost never the first. The whole verification discipline is one question: **what check would catch a forty-foot error?**

1

Known answers first

Before you trust the tool on anything new, run it on a job you already computed by hand. If the tool disagrees, one of you is wrong, and now you have something to investigate. It's a control check: you don't start a traverse by trusting the instrument.

2

Independent checks

Ask the AI to write automated tests — small pieces of code that run the tool against known inputs and confirm the outputs every time anything changes. A test is a check that reruns itself. The discipline is *what* it checks: the math, not the look.

3

Never weaken a failing test

When a test fails, the AI's first instinct is often to loosen it — widen the tolerance, skip the case — because that makes the red go away. The failing test is usually right. Fix the tool, not the test. Put that sentence in your standing instructions.

Every output looked perfect. Every output was wrong.

From the Boundary Tracker build. The tool reads a survey drawing and draws every project boundary on a map. The AI's first approach to flattening curves converted each circular arc to a Bézier curve — the kind graphics software uses — and flattened that. Every map looked perfect.

A Bézier standing in for a circular arc carries a built-in radial error of roughly 0.00027 × the radius, and no tolerance setting can reduce it, because the tool was converging on the wrong curve. A 100-ft radius was off 0.027 ft. A 2,000-ft radius was off **0.545 ft — at every tolerance setting**, from a tenth of a foot to a thousandth.

Nobody would ever have seen it on screen. It was caught because the test wasn't “does the map look right.” The test was “do the computed vertices lie on the true circle” — a known answer, checked with arithmetic. **You define the check. The check tests the math, not the appearance.**

Source: Boundary Tracker validation log, finding V1 (Bézier approximation of circular arcs). Arithmetic you can check in your head: 2,000 × 0.00027 ≈ 0.54 ft.

ASK THE AI TO…

- “Write tests that run this tool against the sample in `/samples` and compare every output value to my hand-checked file. Report any difference larger than 0.005 ft.”
- “Add a test that checks the math directly — [the vertices lie on the true circle / the closure equals the hand computation / the scale factor matches the published county value] — not whether the output looks right.”
- “Keep every test forever. If a test fails, tell me — never loosen a tolerance or skip a case to make it pass.”
- “Run all tests after every change and show me the results before you show me the code.”

FOR THE TINKERERS: REGRESSION TESTS

The name for tests you keep permanently so an old bug can't sneak back in. Once a bug is found and fixed, ask the AI to add a test that would have caught it. The arc test above is one; the Boundary Tracker carries 231 of them. Ask for “a test suite I can run with one command,” and run it before every deploy.

THE FRAME THAT MAKES ALL OF THIS NON-NEGOTIABLE

The tool is unlicensed labor. You are the professional reviewing and approving its work.

When a tech reduces your field data, you don't seal it because you trust the tech; you seal it because you checked the work, or built the process that checks it. A tool the AI built is a tech who has never surveyed, works very fast, and never admits uncertainty. Nothing about your professional responsibility changed — delegation to unlicensed staff is something the profession already knows how to do. Apply it.

Working and safe are two different checks. Ask for the second one — next page.

The security checklist — every line is one sentence to ask the AI

A single HTML file has no server and no door. But the moment a tool runs a local server — the AI's default when you ask for "an app" — it has a door, and anything that can knock can drive it: a web page in another tab, a browser extension, a background worker a site left behind, or anyone on the office Wi-Fi if the server isn't bound to this computer only. Nothing looks wrong. You have to ask.

THE ONE PROMPT — PASTE IT AFTER THE TOOL WORKS, BEFORE ANYONE ELSE USES IT

"Review this tool as if a hostile web page were open in my browser and a stranger were on the office Wi-Fi. List what you find, worst first, and explain each in plain English before you fix it."

Who can reach it?

- ☐ "Make the server listen on this computer only (loopback), so nothing else on the network can reach it." Principle: minimize the attack surface; secure defaults — OWASP Secure Product Design Cheat Sheet. Threat named in OWASP Proactive Controls C8 (DNS rebinding).
- ☐ "Run under my normal user account, never as administrator." Least privilege — OWASP Secure Product Design Cheat Sheet.

What can it touch?

- ☐ "Give the tool one working folder. It must never read or write outside it — not even if a file name or path tells it to (`../`)." OWASP Path Traversal; Input Validation Cheat Sheet ("the client should not be able to specify the file path"). Top 10:2025 A01 Broken Access Control.
- ☐ "Source files are read-only. Copy the job file and work on the copy; fingerprint the original before and after to prove it wasn't touched." Least privilege, as above. The Boundary Tracker shows the fingerprints (SHA-256) to the user.

What does it trust?

- ☐ "Treat every input as hostile — every file dropped in, every field typed, every request that arrives. Validate against an allow-list." OWASP Input Validation Cheat Sheet.
- ☐ "Require something web pages can't supply — a token the tool issues to its own page, or a custom header — before acting on any request." OWASP CSRF Prevention Cheat Sheet (synchronizer token; custom request headers).
- ☐ "Check where each request came from and refuse anything that isn't from the tool's own page." OWASP CSRF Prevention Cheat Sheet, verifying Origin with standard headers ("We recommend blocking").
- ☐ "Apply output encoding, a Content Security Policy, and parameterized queries where they apply — and tell me in plain English where you did." OWASP XSS Prevention, CSP, and Injection Prevention Cheat Sheets; Top 10:2025 A05 Injection.

INSTALLING SKILLS — READ THIS BEFORE YOU DOWNLOAD ONE

A skill is a set of instructions the AI will follow and, often, code it will run — with every permission the AI has. People share them online. **Treat a downloaded skill exactly like a downloaded program.**

1. **Trusted source only.** Watch for look-alike names ("gogle-workspace") — typosquatting is how malicious skills get installed.
2. **Read it before you install it.** It's plain text. If you can't tell what it does, don't install it.
3. **Never paste the "prerequisite" install commands** a skill tells you to run. That's the delivery mechanism.
4. **Ask the AI to list what the skill can touch** — files, network, shell — and reject any skill that needs more than its job requires.

Source: OWASP Agentic Skills Top 10, v1.0 (2026 edition; incubator project — cite as guidance, not an established standard): AST01 Malicious Skills ("Never auto-execute 'Prerequisites' sections without explicit user review"); AST03 Over-Privileged Skills (require a permission manifest); AST05 Untrusted External Instructions ("Prefer inlining over fetching"). owasp.org/www-project-agentic-skills-top-10

"I ONLY VISIT SAFE SITES" IS NOT A CONTROL

You don't control what's already in your browser: an extension you installed two years ago, a background worker a site left behind, a perfectly legitimate site that got hacked last week. Build the tool so it doesn't matter. You don't need to understand the mechanism — you need to ask for the outcome and make the AI explain it.

All cheat sheets: **cheatsheetseries.owasp.org** · OWASP Top 10:2025: **owasp.org/Top10/2025**. Loopback binding and "never paste a credential" are cited here to the least-privilege / attack-surface principles, not as named OWASP recommendations. Honest limit: this is a first line of defense for a tool on one desk. The day a tool sits on a shared server holding client data, it's an IT conversation, not a prompt.

Simplest deployment: one HTML file. No installs. No server.

The simplest deployment there is: one HTML file. Same kind of file as a web page, but it runs from your hard drive, in any browser, no internet. No installs, no server, no admin rights, nothing for IT to object to. Email it or drop it on the shared drive — and deployment includes the ten-minute walkthrough with the people who'll use it. **A tool nobody uses is worse than no tool.**

THE SINGLE-FILE TEST — ONE QUESTION

Can the tool do its whole job with only what the user hands it, and hand the result back? File in, math, file out — a single file will do. It needs a server (a door) for exactly three reasons:

1. It has to reach into your machine on its own — crawling a folder of drawings, say.
2. It has to remember across people or months — that's a database.
3. It has to run unattended, or hold a secret.

The Point Data Converter needs none of those. The Boundary Tracker needs all three. The AI defaults to the server version, so say it up front (prompt 5, page 7).

Every piece of borrowed code is a problem you now own

A dependency is a chunk of someone else's software your tool leans on. It changes, breaks, or vanishes on their schedule. Three habits:

- **Prefer boring, well-worn technology**, and ask for it by name. Bonus: the AI is most reliable with boring technology, because that's where its training runs deepest.
- **Fewer moving parts, fewer surprises.** Every box under your tool can break on somebody else's schedule.
- **A single-file tool has almost no dependencies at all** — the second big reason to start there.

ASK THE AI TO… (WHEN THE TOOL IS DONE, AND EVERY TIME YOU COME BACK TO IT)

- "Run the dependency audit and report anything with a known vulnerability, in plain English."
- "Pin the versions, so the tool only ever installs the exact versions we tested — not whatever came out last night."
- "List every dependency, what it does, and what we'd do if it disappeared."
- "Use only standard, well-supported browser features — nothing that relies on a quirk a browser update could close."

OWASP Vulnerable Dependency Management Cheat Sheet ("perform automated analysis of the dependencies from the birth of the project"; tools: npm audit, OWASP Dependency-Check, Dependency-Track, Trivy, Grype) · OWASP Top 10:2025 A03 Software Supply Chain Failures ("Deliberately choose which version of a dependency you use and upgrade only when there is need"; "Only obtain components from official (trusted) sources… Prefer signed packages").

Three Deploy habits keep you out of trouble

1 · Client data only on terms you've read

Business tiers don't train on your content by default; individual tiers may unless you opt out. Build and test on synthetic data; run client data only on terms you've read. Owners: that's the ten-minute data rule from page 5.

2 · Source files are read-only, always

The tool copies the job file and works on the copy. It never opens the original for write. Make it provable, not promised: fingerprints before and after.

3 · Passwords and permissions stay with you

Never paste a credential into a prompt — the AI only needs the *name* of the setting, not the value; the value lives in a file the AI is told not to read (`.env`). When an agent asks to install something or reach the internet, read what it wants and why. "The AI asked" is not a reason to click yes.

OWASP Secrets Management Cheat Sheet: secrets are never hardcoded in source or configuration, never logged. "Never paste into a prompt" is this handout's extension of that rule.

ADOPTION — THE PART AFTER DEPLOY — DEPLOYMENT ISN'T DONE UNTIL THIS IS

- The ten-minute walkthrough, in person, with the actual people. Then watch one of them use it without you.
- Put the tool where the work already happens — next to the job folder, not in a new place to remember.
- Name one owner and one backup. Write both names in the plain-English explanation.
- Ask for the first three complaints. Each is the next "one change per request."

One file gets you far. Three upgrades take you the rest of the way.

The line between Demo 1 (one file · one task · one desk) and Demo 2 (many drawings · a shared map · several people) is crossed in exactly two ways: when you **chain tools together**, or when the tool needs a **memory that more than one person shares**. Cross it on purpose, not by accident.

All three are "with a little help" territory for a first-timer — the AI can walk you through each, but this is where a second set of eyes earns its keep. Each card: the plain-English meaning, what to ask for, and what to insist on.

1 · A database = shared memory

Plain English: a spreadsheet many tools and people can read and write at once without stepping on each other. **Ask for:** the boring option — SQLite for one machine, Postgres when several people share it. **Insist on:** a one-command backup, and a plain-English description of every table.

2 · Sign-in = who's allowed in

Plain English: who may enter, and what each person may see or change. Matters the instant the tool leaves one laptop — it's the "who can reach it" question from page 9, answered properly. **Ask for:** a well-known sign-in library, never a hand-rolled one; passwords stored only as hashes; the fewest roles that work (usually two: view, edit).

3 · Version history on GitHub = the tool's field book

Plain English: every change recorded, every change reversible, and someone — or the AI, months later — can see how it got here. **Ask for:** "set this folder up under Git with a GitHub remote and show me the three commands I'll use." **The habit:** commit before every AI edit. The undo button is what makes experimenting safe.

Two finished tools, honestly

Point Data Converter — the "you could build this next week" proof

Takes a raw field-point CSV, applies the county surface adjustment factor, cleans point numbers, flags duplicates, exports a processed CSV. One HTML file; zero installs; seven wizard screens. Built in about four hours over two sittings in April 2026, standing instructions first, checked against hand-processed sample pairs before it was trusted. The surveyor still picks the county and confirms the factor — that step is a choice, not a default.

Boundary Tracker — the "and it scales" proof

Project folders are indexed by number, not by location; "what have we surveyed near here?" had no answer except somebody's memory. The tool captures boundary linework from project drawings onto a shared map; every boundary links back to its drawing; exports Shapefile/KML. About 30 logged hours across five weeks of sessions (not a weekend, not months), 231 automated tests, metered AI usage equivalent to roughly $9 per hour of the builder's time. Source drawings are fingerprinted before and after — provably never touched. Zone and grid/ground are deliberately manual.

FURTHER DOWN THE ROAD — FOR REFERENCE, NOT FOR MONDAY

- **Browsers are tightening how public web pages reach local services** (private-network access rules, HTTPS-for-localhost). The specifics are in flux and vary by browser; if a local-server tool suddenly stops opening after a browser update, this is the first thing to ask the AI about.
- **A native desktop app** (built with a framework such as Tauri or Electron) removes the browser door entirely — but a bad dependency then has access to the whole machine, which is the strongest argument for the "boring, few dependencies" rule. A fork in the road you don't need to take first.
- **Containers (Docker)** isolate a tool from the machine it runs on. Real value on a shared server; too heavy for a desk tool.
- **Optional awareness feeds** if a tool becomes load-bearing: the browser vendor's platform-status page, MDN's browser-compatibility tables, CISA's Known Exploited Vulnerabilities catalog. Not this audience's job by default — an IT conversation once a tool holds client data.

What goes wrong in month six — and the habit that prevents each

Every risk comes with the habit that neutralizes it. Four of them: security (page 9), dependencies (page 10), and the two below — the ones where nothing breaks and the firm gets worse anyway.

The tool you built is the tool you now maintain

The trap: a bespoke tool works beautifully, becomes load-bearing, and then depends on one person who "knows how it works" to babysit it. When that person leaves, the tool becomes a liability. AI is both what creates this risk at scale — everyone can build tools now — and what finally makes it manageable: a fresh AI session can pick up a tool it has never seen and maintain it, **if three things exist.**

1 · Keep the "why" written down

A `DECISIONS.md` file: "we chose X because Y." Stops the next person — or the next AI session — from undoing a choice you made on purpose.

2 · Keep the plain-English explanation current

Every time the tool changes, ask for the layman's version of what changed and save it next to the tool. That's the document a successor reads.

3 · Keep the history

Version control: every change recorded and reversible (page 10). Free, and the single most useful habit in this handout.

> **Field notes exist so someone else can retrace the survey. Same rule for tools.**

Automate the friction. Never automate the judgment.

The question to ask of every tool before it ships: *what must our staff actually know how to do — and what is just friction?*

FRICTION — AUTOMATE IT, WITHOUT GUILT	JUDGMENT — THE TOOL MAY SUPPORT IT, NEVER MAKE THE CALL
Reformatting point files	Grid / ground decisions
Retyping deed information into a spreadsheet	Closure analysis
Finding the right drawing in a folder with six FINALs	Boundary resolution; reading a deed and deciding what it means
Renaming photos by point number; checking a plat against a checklist	Choosing the coordinate system and declaring grid-or-ground for a drawing

The tell: someone who can't do it by hand can't QC the tool that does. A firm that automates the judgment doesn't just risk a bad plat — it loses the ability to notice one. The Boundary Tracker could guess the zone and grid/ground of every drawing it reads; it deliberately doesn't — the save button stays off until a person confirms both.

THE PERSONAL VERSION OF THE SAME RULE

learn one thing per session. Pick one thing the AI built and understand it completely. Your ability to QC these tools is what stands between "efficient" and "exposed."

ROI — what owners should actually track

Measure before you build, and again a month after. Keep it to numbers a party chief would believe.

METRIC	HOW TO GET IT
Hours saved per job	Time the task by hand on two jobs before the tool; time it with the tool after. Multiply by jobs per month.
Rework caught or avoided	Count the corrections the tool flags (unknown codes, duplicates, unmapped columns) that used to reach the drafter — or the client.
"Where's that file?" moments	A week's tally before and after. Lost-information time is the biggest hidden cost in most survey offices.
Turnaround on the deliverable	Days from field return to a checked file, before and after.
Cost to build and keep	Hours of your time × your rate, plus the subscription. Then the honest denominator: hours per month to maintain it.
Who else can run it	Number of staff who've done the ten-minute walkthrough. A tool one person can run is a risk, not an asset.

AI-Assisted Development Practices

Working rules for building software with AI coding assistants. Short by design — read it at the start of a session, not once. Pin it above the monitor.

SET UP BEFORE YOU BUILD

- **Start with a plan.** Write the spec first. The model builds exactly what you describe, so decide up front.
- **Maintain `CLAUDE.md` / `agents.md`.** Stack, conventions, off-limits files. It's the briefing every session starts from.
- **Use skills.** Pre-built for common formats; custom ones for domain rules (survey standards, DXF handling) so they apply consistently. Trusted source only · read before installing · never run its "prerequisite" commands unread (page 9; OWASP Agentic Skills Top 10).
- **Match the model to the task.** Cheap/fast for boilerplate and edits; strongest model for architecture, debugging, and security.
- **Keep secrets out of prompts.** Credentials live in `.env` (git-ignored). The model only needs to know the variable name.
- **Prefer boring tech.** Postgres, Python, React. Models are most reliable where training data is deepest.
- **Build modular.** Small files, one job each, clear interfaces between them. The model reasons well about one piece at a time and poorly about a 2,000-line file; a mistake stays contained and a piece can be swapped without touching the rest.

WHILE BUILDING

- **Small, verifiable steps.** One feature or fix per request. Large outputs hide bugs.
- **Commit before every AI edit.** `git checkout` is the undo button.
- **Explain before build.** Ask "how would you approach this?" first. Cheap way to catch wrong assumptions.
- **Give real inputs.** Sample files, actual error text, screenshots — not descriptions.
- **Tests are the referee.** Write them; make the AI run them. "Make the tests pass" beats "looks right."
- **Read the diff.** The summary and the actual change can differ. Skim every changed line.
- **Fix root causes.** Ask "why did this break?" before accepting a patch.
- **Ship a thin slice end to end** before adding breadth. Integration problems surface early and cheap.

OVER THE LONG HAUL

- **Use GitHub for everything.** History, reversibility, and the ability to resume months later.
- **Keep `DECISIONS.md`.** "We chose X because Y." Stops relitigating and stops the AI reversing deliberate choices.
- **Fresh sessions often.** Long chats drift. New session + `CLAUDE.md` = clean context.
- **Second opinion on architecture.** A different model or fresh session critiques the plan.
- **Ask for the plain-English version.** If you can't explain the code, you can't maintain it.
- **Learn one thing per session.** Pick something the AI did and understand it fully.

The same card, in survey terms

SETUP · PLAN

Write the scope before you mobilize. Leave the desk binder for the new hire. Use the SOP sheets — but only ones you'd trust with the keys. Don't put the crew's passwords in the field book. Use the instrument you know. Break the level run into loops that close on their own.

EXECUTE · REVIEW

One task per work order. Back up the job file before every edit. Talk through the boundary before you draft the plat. Hand over the real field file, not a description of it. The check is independent of the person who did the work. Read the actual change, not the summary. Find the cause of the bust, not just the fix. Deliver one complete sheet before you start the whole set.

DEPLOY · ITERATE

The field book stays with the job. Write down why you held the found monument. Start a fresh page for a new job. Have a second surveyor look at the resolution. If you can't explain it to a party chief, you can't seal it. Learn one thing from every job.

The card is the presenter's own working notes, reproduced nearly verbatim; the skills line carries the awareness flag per page 9.

Getting started — pick the path that matches your seat

IF YOU WALKED IN SKEPTICAL

The party chief / project surveyor

1. **Monday morning:** write the one-sentence problem statement for the thing you'd build first (worksheet, page 16). That's the whole assignment.
2. Hand it to a colleague. If they'd need to ask three questions, rewrite it.
3. Fill the spec template on page 6 for it — Problem, Input, Output, Rules, Done-when.
4. Open the chat app you already have and paste prompt 2 (page 7) with your spec. Read what comes back. You haven't built anything yet; you've found out whether the problem is real.

IF YOU ALREADY TINKER

The survey tech / CAD lead who writes LISP or field-code macros

1. **This week:** install one of the two agent tools (below). Let it walk you through the setup.
2. Write the standing-instructions file (page 5). Prompt 1 on page 7 gets you a draft.
3. Build the point reformatter — the training-wheels project. Single file, one input, one output, checked against a job you already did by hand.
4. Put the folder under Git before the second session. Commit before every AI edit.
5. Then one of the three starter projects below — or the first honest answer on page 16.

IF YOU RUN THE FIRM

The principal who thinks in liability and billable hours

1. **Before anyone builds anything:** set the data rule (page 5) — which plan, whether the opt-out is set, what client material is allowed in at all. Ten minutes, once, for everyone.
2. Name the friction/judgment line for your office (page 12) so staff know what's fair game.
3. Measure one task before it's automated (page 12 metrics) so you'll know what the tool was worth.
4. Ask every tool that ships two questions: who maintains it, and who's the backup?

The tools and what they cost

ECOSYSTEM	START HERE	INCLUDES THE AGENT	HEAVY USE	FOR THE FIRM	TRAINING ON YOUR CONTENT
Claude (Anthropic) claude.com/pricing	**Pro — $20/mo** ($17/mo billed annually)	Claude Code (terminal agent) and Claude Cowork (desktop agent for non-developers) are included in Pro and above	Max — from $100/mo (5× Pro usage); $200/mo for 20×	Team — $25/seat/mo ($20 annual); premium seat $100–125	Individual plans: opt-out. Team/Enterprise: "No model training on your content by default."
ChatGPT (OpenAI) chatgpt.com/pricing	**Plus — $20/mo** (Go, $8/mo, has limited Codex access)	Codex (coding agent) is included with Plus and above	Pro — from $100/mo; $200/mo for the top tier	Business — $25/user/mo ($20 annual), 2+ seats	Individual plans: used for training by default, opt-out available. Business/Enterprise/API: not used by default.

Verified against the vendors' own pricing and privacy pages 2026-08-28 and re-checked 2026-09-16 (claude.com/pricing · chatgpt.com/pricing · openai.com/enterprise-privacy). Prices are USD before tax and change at vendor discretion; model version names are deliberately omitted — they churn monthly and add nothing to the decision.

Three questions people ask after this talk

"What's the monthly cost, really?"

Twenty dollars to start, on either ecosystem, and that tier includes the agent. A hundred or two if you go heavy — and you'll know you're heavy because the tool tells you when you hit the limit. The expensive input is your hours, not the subscription; that's what the ROI table on page 12 measures.

"What happens when it breaks in six months?"

If the three things on page 12 exist — the decisions file, the plain-English explanation, the version history — a fresh AI session can pick the tool up cold and fix it, and so can a successor. If they don't exist, the tool depends on one person's memory, which is the same liability as a job folder nobody else can navigate.

"Would you let a client rely on one of these tools?"

Only the way you'd let a client rely on any unlicensed staff member's work: after you checked it against a known answer, with independent tests, and with your review standing behind it. The tool is unlicensed labor (page 8). Nothing about your professional responsibility changed — and nothing needed to.

Three starter projects — each is one file, one input, one output, and checkable by hand

Pick the one closest to a real pain in your office. Each is written in the spec format from page 6, so the "first prompt" line can go straight into the tool after the standing-instructions file exists (page 5). Time estimates assume Setup is done. All three pass the single-file test on page 10.

1 · AN AFTERNOON

Day-rate calculator

Problem: given crew size, equipment, drive time, and hours on site, produce the quoted day rate our office uses. **Input:** five numbers typed in. **Output:** a rate and a one-line breakdown. **Rules:** your rate table, written into the standing instructions. **Done when:** it matches last month's three quotes to the dollar.

FIRST PROMPT

"Build a single-file HTML day-rate calculator using the rate table in CLAUDE.md. Inputs: crew size, equipment class, drive hours, site hours. Show the arithmetic under the result. Before coding, tell me how you'll approach it."

Teaches: the spec, the single file, the known-answer check. **Where people get stuck:** the rate table isn't written down anywhere — which is the point.

2 · A DAY

PNEZD ↔ CSV reformatter

Problem: convert a data-collector export to the column order and code list our template imports, and back. **Input:** one .csv (attach a real one). **Output:** one .csv; unknown codes flagged, never guessed. **Rules:** US survey feet; point numbers unchanged; never touch the original. **Done when:** a job you already imported by hand round-trips with zero differences.

FIRST PROMPT

"Here is a real export (attached) and the hand-corrected file I made from it (attached). Build a single-file HTML tool that turns the first into the second using the code list in codes.csv. Flag any code you can't map. Then write a test that proves the attached pair matches exactly."

Teaches: real inputs, the code list as domain knowledge, tests. **Where people get stuck:** describing the file instead of attaching it.

3 · A WEEKEND

Traverse closure checker

Problem: given a list of bearings and distances, report linear closure, precision ratio, and area. **Input:** a typed or pasted course list. **Output:** closure, ratio, area — and the misclosure by component. **Rules:** US survey feet; bearings in quadrant form; show the arithmetic. **Done when:** it matches three traverses you've already computed, to the hundredth.

FIRST PROMPT

"Build a single-file HTML traverse closure checker. Courses are quadrant bearings (N 45°30'15" E) and distances in US survey feet. Report linear closure, precision ratio, and area, and show latitudes and departures per course. Use these three hand-computed traverses (attached) as tests; never loosen a tolerance to pass."

Teaches: tests that check the math, not the look — and why the judgment (accept or re-run) stays with you. **Where people get stuck:** letting the tool "adjust" the traverse. Don't; that's judgment.

BEFORE IT SHIPS — THE FIVE CHECKS

- ☐ Matches a job I already computed by hand (page 8)
- ☐ Has tests that check the math, and I've seen them run (page 8)
- ☐ Passed the security-review prompt if it has a server; findings explained to me in plain English (page 9)
- ☐ Single file — or I know which of the three reasons it isn't (page 10)
- ☐ Plain-English explanation and DECISIONS file exist; someone besides me can run it (page 12)

WHEN YOU GET STUCK — IN THIS ORDER

1. Restate the requirement in one sentence (prompt 6, page 7).
2. Attach the real file and the real error text.
3. Ask "how would you approach this?" and read the plan before any code.
4. Start a fresh session with the standing-instructions file.
5. Ask a second model, or a second person, to critique the plan.

Don't know what to build? Your pain points do.

You don't need an idea. You need an honest answer to one of these four questions. Every honest answer is a tool candidate. Write in the margins; this page is meant to be used.

1. **Which recurring tasks slow your team down the most — and what makes them tedious or error-prone?**

2. **Where does information routinely get lost, duplicated, or stuck?**

3. **What decisions get delayed because the right information isn't easy to find?**

4. **Where do handoffs fail, and what's usually missing?**

MY FIRST TOOL — THE ONE-SENTENCE PROBLEM

A sentence a new hire could act on without coming back with three questions. Problem first, tool second.

IS IT FRICTION OR JUDGMENT?

Friction — automate it. Judgment — the tool may support it, never make the call.

BUILD-OR-BUY CHECK

Does something we already own solve 80% of this?

None of this replaces your judgment. It removes the boring, repetitive work *around* your judgment, and gives that time back to the work that actually needs a surveyor. Your AI adoption doesn't need to be cutting-edge. It needs to be deliberate and effective — maybe even a little boring.

NOTES FROM THE SESSION

APOLLO LENS SOLUTIONS

John Andricopoulos, RPLS, PMOCP, CAIKMP

Founder, Apollo Lens Solutions LLC — practical innovation for land surveying and AEC firms: custom tools built with AI, the training to build your own, and the QC discipline to trust them.

apollolens.io · **info@apollolens.io** · *Practical innovation. Proven expertise.*