PRACTICAL GUIDEBOOK - 2026

From Friction to Value

A field-tested playbook for finding expensive internal friction, sizing it in euros, and building adopted AI-assisted tools that create measurable value.

AUTHOR

Valentin Angenot

DOMAIN

Strategy - Engineering - AI

REFERENCE CASE

Anonymized public-sector operator

CORE RESULT

98 min to 11 min per permit

BUILT	VALIDATED	READY	VALUE
Working browser tool	Tested with end users	Prepared for adoption	Projected; to be measured after rollout

FOREWORD

Why this guide exists

> *The bottleneck has shifted. In many organizations, the hard part is no longer building an internal tool. It is finding the process worth building for, proving the value, and making the result survive real operations.*

For years, a good internal-tool idea usually died in a backlog. Building it required a team, a budget, a security review, a roadmap slot, and patience. AI-assisted development changes that equation. A capable person with enough domain knowledge can now turn a clear specification into a working tool in days.

That does not make judgment less valuable. It makes judgment more valuable. The scarce work is still human: understanding the process, timing the waste, deciding what matters, designing for maintainability, getting users to adopt the change, and proving that value was actually captured.

This guide is based on a real deployment. After the process had been mapped, measured, and scoped, the build phase took roughly ten working days: I built a browser-based tool with Claude to automate a fragmented permit-preparation process across 24 administrations. The tool reduced preparation time from about 98 minutes to 11 minutes per task. Under conservative assumptions, the administrative saving alone is projected at **€596,881 over ten years**.

Read this as a practical field guide. The specific tool is not the point. The point is the method: find the friction, size it, decide whether to build, design for the long run, build with AI, deploy responsibly, bring users along, and measure the return.

Valentin Angenot, 2026

GUIDE MAP

How to use this guide

FAST PATHS

30 minutes: read the Executive Summary, Part I, Part II, and the business-case template. **To build:** read Parts IV to VII. **To deploy in an organization:** read Parts VIII to XI. **For immediate reuse:** use the templates in Part XII.

EXECUTIVE SUMMARY

The argument and the result

Situation

A public-sector operator decided to bring a field operation in-house rather than continue outsourcing it. The financial logic was attractive, but an administrative layer threatened to consume the margin: each job required a permit, and 24 separate administrations imposed their own forms, platforms, deadlines, contacts, rules, and fees.

Complication

The manual process consumed about 98 minutes of skilled time per permit dossier: finding the right procedure, re-keying data, building a compliant signage plan, preparing documents, drafting emails, and checking authority-specific requirements. The knowledge lived largely in one person's head, creating both a recurring cost and a key-person risk.

Resolution

Once the process was understood and the value case was clear, I built a single browser-based tool in roughly ten working days, working with Claude. The operator fills one form; the tool detects the relevant administration, applies its rules, generates the required documents and emails, and helps the user prepare the signage plan through a visual editor on a real map. Preparation time fell from about 98 minutes to 11 minutes per task.

98 -> 11	-89%	€596,881	24
MINUTES PER PERMIT	OBSERVED PREPARATION REDUCTION	PROJECTED 10-YEAR ADMIN SAVING	PROCEDURES UNIFIED

PRECISION NOTE

The tool has been built, tested with end users, and prepared for adoption. The €596,881 figure is a **projected** ten-year administrative saving, not yet a fully measured ten-year realized result. The realized value should be tracked after rollout through adoption, preparation-time, and rejection-rate indicators.

The transferable method

The tool is incidental. What generalizes is the discipline that produced it: assess the process until friction is visible and timed; express the opportunity in money; decide whether to build, buy, or leave; design for maintainability; build quickly with AI; co-construct for adoption; deploy responsibly; measure what is actually realized; and repeat.

REFERENCE CASE

Proof of the value case

The strongest version of the story is not "AI built an app". It is "a measured process friction was converted into a conservative financial case, then removed through an adopted internal tool".

EXHIBIT 1 **Base-case calculation**

INPUT	VALUE	COMMENT
Manual preparation time	98 minutes	Baseline process before tool support
Tool-assisted preparation time	11 minutes	Observed target after automation of documents, rules, and emails
Time saved per permit	**87 minutes**	98 - 11 minutes; used directly in the base-case calculation
Ten-year volume	5,346 jobs	Projected activity volume over the horizon
Loaded labour rate	€77 / hour	Fully loaded operational cost, not gross salary
Projected administrative saving	**€596,881**	87 / 60 x €77 x 5,346

FORMULA

```
Projected saving = (base-case minutes saved / 60) x loaded hourly rate x job volume
```

EXHIBIT 2 **Sensitivity of 10-year value**

SCENARIO	MINUTES SAVED	JOBS / 10Y	RATE	PROJECTED VALUE
Pessimistic	70	4,000	€70	**€327K**
Conservative base	87	5,346	€77	**€597K**
Central	112	5,346	€77	**€769K**
Optimistic	138	6,000	€80	**€1.10M**

Included in the number

Only direct administrative labour time saved on permit preparation.

Not included in the number

Reduced rejection loops, faster onboarding, lower key-person risk, strategic de-risking of internalization, and improved user experience.

PART I

I

Business Process Assessment

You cannot improve what you have not measured, and you cannot measure what you have not mapped. Before any tool is conceived, the existing process must be made visible, timed, validated, and costed.

1.1 - Why value hides inside processes

Friction is invisible because it is distributed. One tedious task is not a business case. The same task repeated hundreds of times by skilled staff becomes structural cost. It rarely appears as a line item; it is absorbed into salaries and protected by habit.

The more local, fragmented, or person-dependent the process is, the more likely it is to hide value. Undocumented process knowledge is a liability: it creates recurring cost and makes the organization dependent on the few people who know how the work really gets done.

1.2 - Field assessment protocol

A credible baseline is not produced in a single workshop. It is built through interviews, observation, timing, and validation. The aim is to understand the real process, not the process described in a procedure manual.

EXHIBIT 3 **Assessment protocol before any build decision**

STEP	WHAT TO DO	EVIDENCE TO KEEP
Frame the perimeter	Define the start and end of the workflow, the eligible cases, and the cases deliberately excluded from version 1.	Scope note, exclusions, owner validation.
Interview the operators	Ask what takes time, what creates rework, what depends on memory, and what they would never trust a tool to decide alone.	Pain-point list, edge cases, adoption risks.
Observe real cases	Walk through recent or live dossiers. Capture screens, documents, handoffs, decisions, and waiting points.	Annotated process map and sample dossiers.
Time the work	Measure each step with ranges first; use sampled cases when the value case is material.	Time sheet, assumptions, observed variation.
Validate the baseline	Review the mapped process and timings with users and the business owner before pricing the opportunity.	Validated baseline and confidence level.

1.3 - The four-lens diagnostic

EXHIBIT 4 **The four-lens process diagnostic**

01 - MAP	02 - MEASURE	03 - DIAGNOSE	04 - QUANTIFY
Document every step, actor, handoff, input, and output.	Time each step. Use ranges first, then samples for high-stakes baselines.	Classify steps as value-adding, necessary, or waste.	Convert wasted minutes into money using volume and loaded cost.
Output: process map	*Output: time sheet*	*Output: waste register*	*Output: cost of friction*

1.4 - Map the work

Use a swimlane map. Each actor gets a lane: field operator, manager, administrative authority, external provider, or IT system. The point is not design aesthetics; it is to reveal handoffs, waiting time, re-keying, rework loops, and points where only one person knows the procedure.

START
What triggers the task? Which input is needed before work can begin?

ACTORS
Who touches the task, approves it, blocks it, or fixes it after rejection?

RULES
Which rules change by authority, geography, timing, or case type?

OUTPUT
What document, email, decision, or data record proves the task is complete?

1.5 - Measure honestly

Start with ranges, not false precision. Then improve the baseline through structured walk-throughs and sampling. Do not average away complexity too early: conditional steps, rework loops, and exceptions often carry most of the hidden cost.

- **Self-estimate:** fast and useful for orientation, but biased by memory and habit.
- **Structured walk-through:** sit with the user through one real case and time each step.
- **Sampling:** log multiple real cases; use it when the decision depends on the number.

1.6 - Diagnose value versus waste

- **Value-adding:** the beneficiary or organization would recognize the step as useful.
- **Necessary but non-value-adding:** compliance, handoff, or control work that must exist but should be minimized.
- **Pure waste:** searching, reformatting, re-keying, waiting, duplicated communication, and rework after avoidable rejection.

1.7 - Quantify and validate the baseline

> COST-OF-FRICTION FORMULA
>
> ```
> Annual friction cost = (minutes of waste per task / 60) x loaded hourly rate x
> annual task volume
> ```

Use the loaded rate: salary, employer charges, supervision, tools, and overhead. A credible business case can be conservative without pretending skilled labour is free.

BASELINE VALIDATION GATE

Before moving to solution design, the business owner and at least one real user should agree on four items: the mapped workflow is accurate, the timing is plausible, the volume is defensible, and the first automation scope excludes the right edge cases.

PART II

II
Value Discovery & Targeting

A process assessment surfaces many irritants. Resources are finite. This part turns pain points into a prioritized opportunity worth building for.

2.1 - Four forms of value

EXHIBIT 5 **Value taxonomy**

FORM	DESCRIPTION	HOW TO HANDLE
Hard savings	Direct labour time recovered.	Quantify fully.
Risk reduction	Fewer errors, fewer rejections, less dependence on one person.	Price as expected loss when possible; otherwise state qualitatively.
Capacity unlocked	Freed time redeployed to higher-value work or faster onboarding.	Quantify partially and explain assumptions.
Optionality	De-risks a larger initiative or creates reusable internal capability.	Flag explicitly; do not bury it in vague language.

2.2 - Prioritize by impact and feasibility

Start where value is material and feasibility is high. Avoid the trap of choosing the most intellectually impressive problem. The best first AI-built internal tools are often boring, repetitive, and domain-specific.

Start here

High value, high feasibility, clear owner, measurable baseline.

Study later

High value but unclear feasibility, unstable process, or unresolved governance.

Leave alone

Low volume, weak pain, no owner, or mature off-the-shelf solution already available.

2.3 - Stress-test the number

Every business case should answer the uncomfortable question: *under what assumptions does this stop being worth doing?* Vary the minutes saved, volume, labour rate, adoption rate, and maintenance burden. This makes the case more credible, not weaker.

2.4 - The one-page business case

The output of value discovery should be a one-page decision case. A senior decision-maker does not need every observation; they need the recommendation, the prize, the cost to capture it, the risks, and the decision required.

EXHIBIT 6 **One-page business case structure**

BLOCK	QUESTION ANSWERED	CONTENT TO INCLUDE
Recommendation	What should we approve?	Build, buy, or leave; one-sentence rationale.
Problem	What friction are we removing?	Current process, pain point, volume, wasted time, quality or key-person risk.
Value case	What is the prize?	Baseline time, target time, volume, rate, value range.
Cost and effort	What does capture require?	Build cost, deployment effort, maintenance model, owner.
Risks	Why might value not materialize?	Adoption, data quality, rule changes, security, dependencies.
Decision ask	What exactly is needed now?	Approval for pilot, access, owner, budget, or go/no-go.

DECISION QUALITY TEST

The one-page case is ready only when a decision-maker can answer three questions without opening the appendix: what is the problem worth, why is this the right capture path, and what must be true for the value to materialize?

```
Turn this opportunity into a one-page decision case. Put the recommendation first. Use
six blocks: recommendation, problem, value case, cost and effort, risks, and decision
ask. Keep assumptions explicit and move non-critical detail to an appendix.
```

PART III

III
Build, Buy, or Leave

AI-assisted development lowers the cost of building. It does not remove the obligation to choose intelligently.

3.1 - Decision rule

- **Buy** when a mature product solves the process at lower total cost, with acceptable fit and governance.
- **Build with AI** when the problem is local, specific, rule-heavy, and too narrow or fragmented for standard software.
- **Leave it** when the value at stake does not justify the adoption, security, and maintenance burden.

The reference case was a strong build candidate: the rules were local, non-standard, frequently changing, and embedded in operational know-how. The value was high enough, the technical scope was bounded, and a simple client-side architecture could reach users quickly.

3.2 - When not to build

EXHIBIT 7 **Red flags before building**

RED FLAG	WHY IT MATTERS	WHAT TO DO INSTEAD
No clear process owner	No one will maintain or defend the tool.	Find ownership before building.
Low volume or low pain	The tool may be elegant but economically irrelevant.	Automate only if build effort is tiny.
Unstable process	You may encode rules that change before adoption.	Stabilize the process or design configuration first.
Sensitive data without governance	Value can be destroyed by privacy or security failure.	Involve IT, security, legal, or the DPO early.
Users not involved	Adoption risk can erase the value case.	Pilot with champions before scaling.
Off-the-shelf tool solves 80%	Custom build may create unnecessary maintenance burden.	Buy or configure rather than build.

3.3 - The decision should include adoption

A tool used by 40% of eligible users captures at most 40% of the projected return. Adoption is not a soft topic after the business case; it is a value driver inside the business case.

PART IV

IV
Designing for the Long Run

A tool that works on launch day but cannot be maintained six months later destroys more value than it creates. Good design is mostly decided before the first line of code.

4.1 - Optimize for total cost of ownership

The build is a one-off. Operations last years. Design for the years: updates, rule changes, handover, deployment, documentation, access, data protection, and recovery after mistakes.

4.2 - Separate data from logic

The single highest-leverage design choice is to externalize everything that changes. In the reference tool, the 24 rule sets live in a plain data file, separate from the application code. A non-developer can update a contact, deadline, or required document without touching the core logic.

4.3 - Choose the persistence layer deliberately

JSON is excellent for stable configuration: local rules, contacts, fees, deadlines, templates, and examples. A database becomes relevant when the tool must store shared operational records, support multiple users, preserve history, generate reporting, enforce permissions, or integrate with existing systems. The mistake is not choosing JSON; the mistake is choosing it by default when the use case has become transactional.

EXHIBIT 8 **Persistence decision guide**

OPTION	USE IT FOR	AVOID IT WHEN
Static JSON / CSV	Reference rules, configurable templates, lists, thresholds, fees, deadlines.	Users must create, update, share, or audit records over time.
Browser storage / IndexedDB	Drafts, offline work, local history, autosave on one device.	The organization needs a shared source of truth or centralized reporting.
Managed database + API	Shared submissions, user accounts, audit trail, dashboards, status tracking.	The first version can deliver value with static configuration only.
Core-system integration	Authoritative records already live in ERP, GIS, CRM, ticketing, or permitting systems.	A small standalone tool would duplicate official data and create reconciliation work.

Rule of thumb: keep rules in JSON when they are configuration; use a database when users generate records that must be shared, governed, queried, or audited. A strong version 1 can start with JSON and still be designed so a database can be added later without rewriting the entire tool.

4.4 - Keep modules navigable

AI coding can easily produce one giant file. It works today; it becomes unmaintainable tomorrow. Decide the module structure before building.

EXHIBIT 9 **Example module boundaries**

FILE	RESPONSIBILITY
`app.js`	Application state, navigation, event orchestration.
`forms.js`	Form schema, dynamic fields, serialization.
`rules.js`	Administration detection, rule resolution, fees, deadlines.
`validation.js`	Required fields, date checks, coherence controls.
`editor.js`	Visual signage editor and map interaction.
`documents.js`	Document generation and annex formatting.
`emails.js`	Recipients, subjects, body templates, attachments.
`storage.js`	Autosave, local history, browser persistence.
`database.js` / `api.js`	Optional later layer for shared records, authentication, status history, and reporting.
`communes.json`	Configurable administration-specific rules, not transactional records.

4.5 - Security is a design constraint

Security is not a final polish step. It shapes architecture. A client-side tool can reduce data transfer, but it does not magically make data safe. You still need to understand what data is handled, what leaves the browser, who can access the tool, and how updates are governed.

CRITICAL DEPLOYMENT CAVEAT

Free public hosting is appropriate for demos and non-sensitive static tools. For internal operational tools, validate repository visibility, access control, data exposure, ownership, and retention with the relevant IT or security owner before deployment.

4.6 - Pre-build checklist

- ☐ Everything that changes is outside the core code.
- ☐ The module structure is defined before coding starts.
- ☐ The tool runs where the user actually works.

- ☐ Inputs are validated before expensive downstream errors occur.
- ☐ External failures degrade gracefully and do not block the user unnecessarily.
- ☐ No secrets are placed in client-side code.
- ☐ Sensitive or personal data handling is documented and validated.
- ☐ A non-developer can perform routine configuration updates.
- ☐ The persistence decision is explicit: JSON/config, local browser storage, database, or integration.

PART V

V
Building with Claude

The leverage comes from a clean division of labour: human judgment on what is correct and valuable; AI throughput on implementation, refactoring, and iteration.

5.1 - The collaboration model

AI did not replace domain expertise. It collapsed the distance between a precise specification and working software. In the reference case, the scarce input was not syntax; it was understanding the field process, the administrative rules, the edge cases, and what operators would actually use.

EXHIBIT 10 **Division of labour**

HUMAN - JUDGMENT	CLAUDE - THROUGHPUT
Domain research and process understanding.	Specification to first working code.
Definition of correctness and edge cases.	Architecture options and trade-offs.
Stakeholder alignment and adoption design.	Boilerplate, document generation, refactoring.
Final validation and accountability.	Debugging support and fast iteration.

5.2 - Prompting as delegation

A good prompt behaves like a good delegation note: it provides context, defines the task, states constraints, describes what "done" means, and protects what already works.

1. Lead with context before the request.
2. State constraints early: framework, dependencies, data, device, connectivity.
3. Ask for architecture or schema before code when decisions have long-term consequences.
4. Build in vertical slices, not huge horizontal layers.
5. Constrain the edit surface: name the file, function, and behavior to preserve.
6. Test against real cases after each slice and convert defects into regression tests.

5.3 - Worked prompt sequence

The sequence below is deliberately stricter than a normal chat. It keeps Claude focused, reduces accidental rewrites, and makes every build loop testable.

Pass 0 - Project contract

```
You are helping me build an internal browser-based tool after the process has already
been mapped and sized. Users are field operators, not developers. The first version must
run client-side where possible, remain usable with poor connectivity, and keep changing
authority rules outside the core code. Decide explicitly whether version 1 needs only
JSON/config and local drafts, or whether a database/API is justified. Do not write code
yet. First, restate the objective, list the constraints, identify the main risks, and ask
only the questions that would block a safe first build.
```

Pass 1 - Architecture before code

```
Propose the simplest architecture for version 1. Include: file structure, module
responsibilities, data model, configurable rule file, validation layer, document-
generation layer, persistence choice (JSON/local storage/IndexedDB/database/API), and
deployment assumptions. Compare this against one more complex alternative, then recommend
the minimum viable architecture. Do not generate implementation code yet.
```

Pass 2 - Data model first

```
Design the data model in two layers. Layer 1: JSON/config for authority-specific rules:
authority name, location matching, contacts, deadlines, fees, required documents,
templates, special conditions, and versioning. Layer 2: optional database entities if the
tool must store shared submissions, statuses, users, comments, audit trail, or reporting
metrics. Show the JSON schema, then list the database tables only if they are justified.
Do not write application code yet.
```

Pass 3 - First vertical slice

```
Build the smallest working vertical slice: user enters address, dates, and authority; the
app validates required fields; it loads the matching authority rules from JSON/config; it
saves a draft locally or through the chosen persistence layer; it generates one preview
document and one email body. No extra features. Keep the code modular and readable.
Output only the files needed for this slice.
```

Pass 4 - Harden against real failure modes

```
Harden the slice for real operations. If location detection fails, the user can select
the authority manually. If a required field is missing, show a clear message next to the
field. If local storage fails, warn the user without blocking document generation.
Preserve all existing behavior and change only the relevant modules.
```

Pass 5 - Real-case correction

```
I tested the tool on a past case. Expected output: [expected]. Actual output: [actual].
Identify the likely root cause, propose the smallest safe fix, and give me a regression
test that would catch this error in the future. Do not refactor unrelated code.
```

PART VI

VI
Advanced Prompting & Token Strategy

AI-assisted building stays cheap and effective only if you manage context, specificity, edit surface, and output size.

6.1 - Twelve practical prompting patterns

EXHIBIT 11 **Prompting patterns for coding**

PATTERN	PURPOSE	INSTRUCTION SHAPE
Project contract	Aligns objective and constraints before code.	"Restate the goal, constraints, risks, and blockers before coding."
Schema first	Prevents expensive rewrites.	"Propose the config schema and database entities. Do not code yet."
Persistence decision	Avoids overbuilding or underbuilding storage.	"Compare JSON, local storage, IndexedDB, DB + API, and core-system integration."
Architecture options	Forces trade-offs before lock-in.	"Compare 3 options on maintainability, security, deployment, TCO."
Vertical slice	Keeps each loop demonstrable.	"Build input -> validation -> logic -> one output. Nothing else."
Edit boundary	Protects working code.	"Modify only [function/file]. Preserve existing behavior."
Diff output	Reduces tokens and review effort.	"Show only changed functions and a short explanation."
Failure mode	Turns vague hardening into operational behavior.	"If [failure] happens, show [message], allow [fallback]."
Real-case test	Anchors correctness in reality.	"Expected [x], actual [y]. Root cause, minimal fix, regression test."
Self-audit	Catches drift after many turns.	"List functions, callers, dependencies, and boundary violations."
Security review	Surfaces risks AI may miss.	"Review for secrets, XSS, storage, dependencies, data leakage."
Handover	Turns code into an ownable asset.	"Create file map, config guide, tests, deployment, known limits."
Business challenge	Stress-tests the case, not only the code.	"Act as CFO/security/user and attack the weakest assumption."

6.2 - Token economics

Tokens are the unit of AI consumption. Input tokens include the prompt, pasted code, uploaded context, and conversation history. Output tokens include everything the model writes back. The hidden cost is not only price; it is stale context. Long conversations accumulate abandoned attempts, old decisions, and irrelevant debugging trails.

6.3 - Context reset protocol

Do not restart from a blank chat when the conversation drifts. Restart from a checkpoint: latest files, current architecture, decisions that must not be reversed, known bugs, and the next edit. The goal is to remove stale history without losing the true state of the tool.

```
Checkpoint this project in under 300 words. Include: current objective, file map, data
schema, completed features, unresolved bugs, constraints, decisions that must not be
reversed, and the next safest edit. Do not propose new features.
```

6.4 - Eight token optimization techniques

1. Be concise; do not re-explain stable context unnecessarily.
2. Paste only the relevant function or file when possible.
3. Ask for diffs, not full rewrites.
4. Use structured outputs: JSON, tables, or code only.
5. Set length limits: "under 50 lines", "one paragraph", "only changed functions".
6. Start fresh from a checkpoint when the conversation drifts.
7. Store stable architecture rules in a project or handover note.
8. Use lower-cost models for formatting and higher-reasoning models for architecture.

6.5 - Cost benchmark

In the reference case, the direct AI cost was economically negligible compared with the projected value. The expensive input was not the AI usage; it was the human judgment before and during the build: process diagnosis, specification, testing, and adoption work. The ten-day figure refers to the focused build sprint once the opportunity was already understood.

PART VII

VII
Understanding the AI Partner

You do not need to understand an engine to drive a car, but knowing a few mechanics makes you a safer and faster driver.

7.1 - What an LLM is in this workflow

A large language model processes text and other inputs, then generates likely useful outputs. It can transform a specification into code, suggest architecture, explain trade-offs, draft documentation, and help debug. It does not guarantee truth, correctness, security, or current external knowledge unless it is connected to reliable tools and reviewed.

7.2 - Model tiers without dating the guide

EXHIBIT 12 **Generic model selection logic**

TIER	BEST FOR	USE WHEN
Fast / low-cost	Formatting, boilerplate, extraction, simple transformations.	The task is clear and low-risk.
Balanced coding	Most implementation, debugging, document generation, refactoring.	You need quality and speed.
High reasoning	Architecture, security-sensitive choices, ambiguous design trade-offs.	The decision has long-term consequences.

Model names, context limits, and pricing change. Verify current provider documentation before making cost or architecture decisions.

7.3 - Strengths and weaknesses

Strengths

- Turning clear specs into code.
- Generating tedious structures quickly.
- Explaining trade-offs.
- Refactoring when boundaries are explicit.

Weaknesses

- Plausible but wrong output.
- Context drift in long sessions.
- Security blind spots.
- Overfitting to the user's premise.

The mitigation is simple but non-negotiable: human review on critical paths and testing against real cases.

PART VIII

VIII
Co-Construction & Adoption

The graveyard of internal tools is full of technically correct software that nobody used. Adoption is not a launch event; it is a design input.

8.1 - Tools fail on adoption

The hardest problem is rarely making the tool work. It is making users trust it, understand it, and prefer it to the old way. Co-construction reduces that risk because users recognize their own pain and their own feedback in the final product.

8.2 - Adoption levers across the build

EXHIBIT 13 **Adoption levers**

PHASE	CO-CONSTRUCTION ACTION	ADOPTION EFFECT
Assessment	Users validate the process map.	Problem ownership.
Design	Users review the proposed workflow before code.	Early objections surface.
Build	Users see working slices after each loop.	Solution ownership.
Pilot	A small team uses it on real work.	Champions emerge.
Rollout	Champions train peers.	Peer credibility beats mandate.

8.3 - Documentation that outlives the builder

> *Knowledge in one head is a liability. Knowledge encoded in a system and a handover document is an asset.*

After the tool works, use the same AI assistant to help draft a technical handover pack. The builder must still validate it, but AI reduces the cost of documentation that is often skipped when projects move fast.

EXHIBIT 14

Handover pack

DOCUMENT	PURPOSE	CONTENTS
Architecture note	Explain the app structure.	File map, modules, data flow, dependencies.
Configuration guide	Let non-developers update rules.	Data schema, examples, safe-edit rules.
Maintenance manual	Support future fixes.	Common changes, tests, known limits.
Deployment guide	Make redeployment possible.	Hosting steps, rollback, ownership.

PART IX

IX
Deployment & Operations

A tool nobody can reach creates no value. A tool deployed without ownership can create risk. Deployment is both access and governance.

9.1 - Deployment options

EXHIBIT 15 **Deployment spectrum**

OPTION	COST	BEST FOR	CAVEAT
GitHub Pages	Free	Static demos, non-sensitive client-side tools.	Public by default unless repository/ access model is controlled.
Netlify / Vercel / Cloudflare Pages	Free tiers	Static tools, CI/CD, serverless extensions.	Validate data exposure and account ownership.
Managed DB + serverless API	Low to moderate	Shared records, auth, status tracking, dashboards.	Requires access control, backups, data model ownership.
Internal SharePoint / intranet	Existing	Regulated internal environments.	May require IT packaging and access control.
AWS / Azure / GCP static hosting	Low to moderate	Enterprise-grade access, logging, governance.	Requires cloud ownership and configuration.
Docker / managed app platform	Moderate	Backend, database, authentication, integrations.	Higher operational burden.

9.2 - When a database becomes worth it

A database is not a badge of seriousness; it is an operational commitment. Add one when the product has data that must be shared, secured, queried, audited, or integrated. Until then, a static configuration file plus local drafts may be safer, cheaper, and easier to maintain.

EXHIBIT 16 **Database maturity ladder**

STAGE	PERSISTENCE MODEL	TYPICAL TRIGGER
Configuration only	Static JSON / CSV, versioned with the code.	Rules change, but users do not need shared operational records.
Local working memory	localStorage or IndexedDB for drafts and offline history.	Operators need autosave or offline continuity on one device.
Shared operational database	Managed database behind a small API with authentication.	Multiple users need shared submissions, statuses, comments, and reporting.
System of record	Integration with approved enterprise systems or authoritative databases.	The tool must write or read official operational records.

The design question is therefore not "JSON or database?" but "which data is configuration, which data is a record, and which system should be authoritative?"

9.3 - Operations maturity ladder

EXHIBIT 17 **Operations maturity**

LEVEL	STAGE	WHAT IT ADDS
0	Live	Reachable, basic validation, data separated from code.
1	Maintainable	Versioned data, documented update path, test checklist.
2	Resilient	Graceful degradation, offline fallback, internalized critical assets.
3	Integrated	Backend, authentication, links to core systems.
4	Governed	Monitoring, access control, audit trail, formal ownership.

9.4 - Minimum operating model

- ☐ Named business owner.
- ☐ Named technical maintainer or escalation path.
- ☐ Clear update process for changing rules.
- ☐ Basic version control.
- ☐ Test checklist after every change.
- ☐ Documented deployment location and rollback procedure.
- ☐ If a database exists: owner, backup, access model, retention, and migration path are documented.

PART X

X

Measuring Realized Value

A projected return is a promise. A realized return is a result. The difference is measurement.

10.1 - The benefits gap

Two things usually separate projected and realized value: adoption shortfall and assumption drift. The tool may work, but not everyone uses it. The baseline may be wrong. Volumes may change. Savings may be lower than expected. None of this is visible unless the tool and the rollout are instrumented.

10.2 - Benefits-realization indicators

EXHIBIT 18 **Measurement dashboard**

TYPE	METRIC	WHY IT MATTERS
Leading	% of jobs processed through the tool.	Adoption is the largest driver of captured value.
Leading	Active users / eligible users.	Shows reach across the team.
Leading	Rejected-application rate.	Captures quality improvement beyond time saved.
Lagging	Average preparation time per job.	Measures the core business-case assumption.
Lagging	Cumulative hours saved to euros realized.	Compares actual value with projected value.
Lagging	Onboarding time for new users.	Tests whether encoded knowledge reduced key-person risk.

10.3 - Realized-value formula

ACTUAL CAPTURED SAVING

```
Realized value = actual jobs processed x actual minutes saved / 60 x loaded hourly rate x adoption-adjustment factor - maintenance cost
```

Measurement turns the first tool from a nice story into an evidence base for the next tool.

PART XI

XI
Scaling to a Portfolio

One tool is a project. A repeatable ability to find friction and remove it is a capability.

11.1 - From one win to a capability

The first tool's most valuable output is proof that the organization can identify friction, build responsibly, adopt the result, and measure the return. Once patterns exist, marginal cost falls: architecture templates, prompt libraries, data schemas, deployment paths, and governance routines can be reused.

11.2 - Portfolio sequencing

Wave 1 - Prove

High-value, low-risk, certain-to-succeed tools. Prioritize credibility.

Wave 2 - Build

Reuse patterns, train champions, reduce marginal cost.

Wave 3 - Extend

Address harder problems once governance and trust exist.

11.3 - Governance guardrails

EXHIBIT 19 **Portfolio guardrails**

DOMAIN	GUARDRAIL
Data and privacy	Classify data, document flows, involve DPO/security when required.
Human oversight	AI-generated code is reviewed before production.
Ownership	Every tool has a named business owner and maintainer.
Consistency	Shared patterns for data separation, deployment, testing, documentation.
Transparency	Disclose where AI was used; keep the human accountable.

PART XII

XII
Templates & Playbook

The method, distilled into reusable prompts, checklists, and a sprint plan.

12.1 - Ten-day build sprint

IMPORTANT SCOPE NOTE

This is not a ten-day path from a blank page to a finished transformation. The ten days refer to the **build sprint**, after the process has already been mapped, timed, sized, and selected. Proper process mapping usually requires interviews, real-case observation, validation with users, and sometimes several iterations.

EXHIBIT 20 **Preconditions before the 10-day build sprint**

INPUT REQUIRED	WHAT MUST ALREADY BE CLEAR	WHY IT MATTERS
Process map	Steps, actors, handoffs, exceptions, and authority touchpoints.	The tool cannot automate a process nobody has understood.
Measured baseline	Current preparation time, volume, rejection/rework patterns, and loaded labour rate.	The value case depends on a defensible before/after baseline.
Selected scope	The first workflow slice to automate and the edge cases excluded from version 1.	A clear boundary prevents a fast build from becoming an endless build.
User owner	The person or team who validates outputs, tests real cases, and owns adoption.	Without ownership, the tool remains a demo.
Data, database, and security constraints	What data is handled, which data is configuration, which data is a record, where it can be stored, and whether IT/DPO review is required.	Persistence and deployment choices depend on governance, not only convenience.

EXHIBIT 21 **A practical 10-day build sprint once the opportunity is scoped**

DAY	BUILD OBJECTIVE	OUTPUT
1	Translate the validated scope into architecture.	Module map, persistence decision, data model, file structure, constraints, definition of done.
2	Build the first vertical slice.	One usable flow: form input -> validation -> saved state -> one generated output.
3	Externalize rules and design storage.	Editable JSON/config for rules; database/API schema only if shared records, statuses, reporting, or audit trail are needed.
4	Generate the core documents and emails.	First complete permit pack generated from the form and rules.
5	Build the visual or domain-specific editor.	Purpose-built interface for the highest-friction expert task.
6	Harden the workflow.	Validation, manual fallbacks, error messages, offline/poor-connectivity behavior.
7	Test against real or past cases.	Bug list, corrected outputs, edge-case decisions, known limitations.
8	Run the user feedback loop.	Workflow simplifications, adoption notes, training points, champion feedback.
9	Prepare deployment and handover.	Hosting path, configuration guide, database/access notes if relevant, rollback/update procedure.
10	Pilot readiness and measurement setup.	Go-live checklist, adoption metrics, time-saved tracker, realized-value dashboard.

12.2 - Build brief template

```
Context: I am building [tool] for [user] who [situation / constraint].
Goal: [the one outcome that matters].
Pre-work already completed: [process map / baseline / selected scope / owner].
Constraints: [framework, deployment, dependencies, device, data rules, security rules].
Persistence decision: [static JSON / local browser storage / IndexedDB / managed database
+ API / enterprise integration].
Scope now: [the single vertical slice to build first].
Definition of done: [what correct looks like, including edge cases].
Build only the slice above. Ask questions if anything is ambiguous.
```

12.3 - Detailed prompt library

The prompts below are written to reduce ambiguity, protect working code, and force the model to think before implementing. They are intentionally operational.

EXHIBIT 22 **Reusable prompts - strategy and architecture**

MOMENT	PROMPT SHAPE
Start from scoped opportunity	"The process is already mapped and sized. First build scope: [scope]. Users: [users]. Constraints: [constraints]. Before coding, restate the objective, assumptions, risks, and missing information. Then propose the simplest version 1 architecture."
Choose architecture	"Give me three architecture options. Compare maintainability, security, deployment effort, offline behavior, data exposure, ownership, and total cost of ownership. Recommend the minimum viable option and explain what would make you choose another."
Design data model	"Separate configuration from records. For configuration, design a JSON schema for rules that vary by [entity]. For records, propose database entities only if users need shared submissions, statuses, audit trail, or reporting. Include examples, validation rules, versioning, and ownership. Do not code yet."
Choose persistence	"For this tool, compare static JSON, localStorage, IndexedDB, managed database + API, and enterprise-system integration. Score each on maintainability, offline use, multi-user needs, auditability, reporting, security, and TCO. Recommend the simplest safe option."
Design database schema	"If a database is justified, propose a minimal schema: tables, primary keys, relationships, indexes, access rules, retention needs, audit fields, and migration path from the JSON/local version. Keep it as small as possible."
Build vertical slice	"Build the smallest testable slice: [input] -> [validation] -> [rule lookup] -> [output]. No extra features, no speculative abstractions. Keep modules separate and name exactly which files you create or change."
Control module boundary	"This module is responsible only for [responsibility]. It must not touch [forbidden dependency]. It should export only [functions]. If the request requires another module, stop and say so before coding."
Add feature safely	"Add [feature]. Preserve all existing behavior. Change only the necessary functions. Return: changed code, files touched, assumptions, and one regression test."

EXHIBIT 23 **Reusable prompts - build, test, and handover**

MOMENT	PROMPT SHAPE
Fix precise issue	"On a real case, actual behavior was [actual]. Expected behavior is [expected]. Identify the likely root cause, propose the smallest safe fix, and do not refactor unrelated code."
Harden failure mode	"If [service/input/dependency] fails, do not block the user. Show [message], allow [manual fallback], preserve entered data, and log or expose enough detail for troubleshooting."
Reduce token waste	"Output only the changed function or minimal diff. No full-file rewrite. Add a two-bullet explanation: what changed and how to test it."
Audit after drift	"Audit the current file. List every function, responsibility, callers, dependencies, duplicated logic, and module-boundary violations. Do not change code."
Test against reality	"Here is a past case, expected output, and current tool output. Compare them field by field, identify discrepancies, rank them by severity, and propose the smallest fixes."
Prepare handover	"Create a maintainer handover: file map, module responsibilities, data schema, configurable fields, safe-edit procedure, deployment steps, rollback, tests after change, known limitations, and business-owner summary."
Security review	"Review this tool for exposed secrets, XSS, unsafe dependencies, data leakage, localStorage sensitivity, access-control assumptions, and prompt-injection risk. Prioritize fixes by severity and effort."
Business-case challenge	"Act as a skeptical CFO. Challenge the value case. Identify weak assumptions, missing evidence, adoption risks, maintenance costs, and the minimum proof needed to approve rollout."

12.4 - Prompt packs for common build moments

Project checkpoint

```
Create a project checkpoint that can be pasted into a fresh conversation. Keep it under
300 words. Include: business objective, users, current architecture, files and
responsibilities, data schema, completed features, unresolved bugs, security/data
constraints, deployment assumptions, decisions that must not be reversed, and the next
safest edit.
```

Fresh conversation handover

```
You are continuing an existing internal-tool build. Use the checkpoint and current file
below as the source of truth. Respect the architecture and constraints. Modify only
[target file/function]. The requested change is [change]. Preserve existing behavior
unless explicitly told otherwise. Output only the minimal code diff, assumptions, and
tests to run.
```

Real-case validation

```
I am testing the tool on a real past case. Context: [case context]. Expected output: [expected]. Actual output: [actual]. Compare field by field, identify the likely root cause, propose the smallest safe fix, and write one regression test that would catch this error next time.
```

Non-technical user feedback

```
Rewrite this feature explanation for a non-technical operator. Use operational language, not software language. Explain: what the user enters, what the tool does automatically, what the user must verify, what warning signs to watch for, and what to do if something looks wrong.
```

Deployment decision

```
Given this tool handles [data type], has [number/type of users], needs [access constraints], and will be maintained by [owner], compare GitHub Pages, internal SharePoint/intranet, Cloudflare Pages, a managed database + serverless API, and a small authenticated backend. Recommend the safest minimum viable deployment and list the conditions that would force a more governed option.
```

Database decision

```
Decide whether this tool needs a database. Facts: users = [users], records = [records created], sharing needs = [sharing], offline needs = [offline], reporting needs = [reporting], audit/security needs = [audit/security], integration needs = [systems]. Compare JSON/config, local browser storage, IndexedDB, managed database + API, and enterprise integration. Recommend the simplest safe option and list the trigger that would justify moving to the next level.
```

Migration-ready data model

```
Design this version so it can start with JSON/local storage but migrate to a database later. Define stable IDs, record shape, versioning, timestamps, audit fields, validation rules, and import/export format. Explain what must not be hard-coded if future database migration is likely.
```

Post-build security pass

```
Perform a security and maintainability review before pilot. Check: exposed secrets, XSS, document injection, dependency risk, localStorage sensitivity, data flows, access assumptions, manual fallback behavior, and whether a non-developer can safely update rules. Return a prioritized fix list: critical, important, nice-to-have.
```

12.5 - Security checklist

- ☐ No API keys, credentials, or secrets in client-side code.
- ☐ Content Security Policy considered and tested for hosted deployments.
- ☐ Known vulnerability checks performed where dependencies exist.
- ☐ Personal or sensitive data not stored in localStorage unless explicitly governed.
- ☐ Database access, backups, retention, and ownership documented when a database is used.
- ☐ If user input touches an AI model, prompt-injection safeguards are included.
- ☐ User input sanitized before rendering or document injection.
- ☐ Dependencies verified as legitimate, maintained, and necessary.
- ☐ Data flows documented: what leaves the browser, to whom, and why.
- ☐ AI-generated code reviewed on critical paths.
- ☐ HTTPS enforced for hosted deployments.

CLOSING

The method in one page

EXHIBIT 24 **From friction to value**

STEP	QUESTION	OUTPUT
1. Assess	Where does work leak time, quality, or resilience?	Process map and friction register.
2. Size	Is the pain worth solving?	Conservative value case.
3. Decide	Build, buy, or leave?	One-page decision case.
4. Design	Can this survive real operations?	Architecture, persistence model, data model, security constraints.
5. Build	Can AI turn the specification into a tool fast?	Working vertical slices.
6. Adopt	Will users actually use it?	Pilot, champions, training, handover.
7. Deploy	Can the organization run it safely?	Hosting, database/access model, ownership, update path.
8. Measure	Did the projected return become real?	Benefits-realization dashboard.
9. Scale	Can this become a capability?	Portfolio pipeline and guardrails.

> *The barrier to six-figure value inside many organizations is no longer engineering capacity. It is the willingness to look closely at a tedious process, measure it honestly, use AI to remove the friction, bring the team along, and prove the return. That work is within reach of one capable person, and it compounds.*