

From Idea to Validated Software

A free workflow sheet: how I work, and how I validate what I build

Software should never fail a person in silence.

This is the workflow I use to take an idea to working software, with AI inside the process and validation built in from the start. It is written so you can use it on your own projects. Take the stages, the loop and the checklists, and keep whatever helps.

The rule underneath it all. A prototype can be incomplete, but it must never be wrong about the idea. So I write down what the idea must do before anything is built, and I judge what gets built against that record, not against “it runs”.

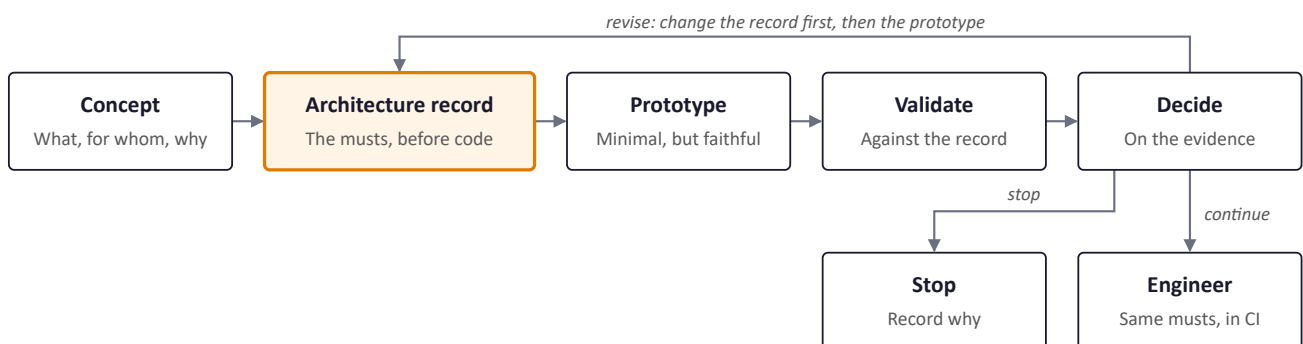
1 • Five stages, each leaving a record

Every piece of work moves through the same five stages. Each one leaves something you can point to later, so every decision stays traceable.

Stage	What happens	What it leaves behind	A lightweight version
Plan	Work is broken into tracked tickets, each tied to an epic and a clear outcome.	A board that shows what is moving, and why.	A text file with one outcome per item.
Decide	Significant choices become Architecture Decision Records, with the options considered and the trade-offs.	The reasoning, not just the result.	One short markdown file per decision, kept in the repo.
Build	Source code, merge requests and CI runs in GitLab. Testing and validation happen as part of building, and CI enforces them on every change.	A history of what changed and why.	Any git host with a pipeline, even one that runs a single check.
Explain	The technical story and the lessons learned, written up as public articles.	Thinking that others can follow and question.	A few paragraphs in the project's README or docs.
Deliver	A case study on my site, ready to review in one place.	A finished result someone else can check.	A release note or a one-page summary.

The tools are interchangeable. What matters is that each stage has one job and leaves a record you can trace.

2 • The loop inside every build



Revising always starts with the record, then the prototype, so the two never drift apart. AI works inside every box. The judgment, the design and the responsibility for what ships stay with a person.

3 · A brief first, then the musts

Start with a one-page brief: the problem, who uses the thing, who is affected by it without using it, the outcome that must change for them, the constraints, the assumptions you have not checked, and what is out of scope. If you cannot fill it in, the idea is not ready to build.

Then write the architecture record. It defines what a prototype must do to count as the idea, it is written before any code, and it constrains everything built after it. It is never documentation written afterward to describe whatever happened to get built.

Components: the parts, and what each one is responsible for.

Flow: what moves between the parts, and where it comes from.

States: the allowed transitions, and the ones that must be impossible.

Musts: numbered rules (M1, M2 and so on), each written so it can be checked.

Deterministic or judgment: what must give the same answer every time, and where judgment is allowed.

Contracts: the exact shape of data at every boundary, taken from real documentation.

Not in the prototype: what is left out on purpose, so no one mistakes an omission for a bug.

Significant choices get a short decision record: the context, the options considered, the decision and its consequences. One page is plenty. What matters is that the reasoning survives.

4 · Prototype: minimal, but faithful

A prototype is the smallest build that meets every must. It can leave out a great deal, but it cannot fake what makes the idea the idea.

It may leave out	It must never fake
Scale, speed, polish and secondary features	The core behavior the idea depends on
Accounts, deployment and production hosting	The states and rules in the record
Real personal data (sample values are enough)	The shape of real data and real API responses
Hardening beyond the musts	Honest failure: saying so when it cannot do something

Working with AI

1. Give it the architecture record as its specification, at the start of every session.
2. Build one must at a time, and commit after each, so every change is traceable.
3. Ask it to list every assumption it made, then check each one against the record.
4. Reject anything it adds that the record does not ask for. If it is genuinely useful, it goes into the record first.

Prompts that help

Brief: "Ask me the ten questions you would need answered before designing this. Do not suggest a solution yet."

Record: "Here are my musts. Which are ambiguous, which contradict each other, and what is missing?"

Prototype: "Implement M1 only. Add nothing that is not in the record, and list every assumption you made."

Validate: "For each must, write test cases that would fail if it were not met, including missing and malformed input."

This is also the answer to generic output and invented integrations. A record that fixes the stack and the musts leaves the model no gaps to fill with its defaults, and a contract taken from the platform's real documentation turns a guessed API into something you can check.

5 · Validation engineering

Validation is not a phase at the end. It is a set of boundaries, each placed where a bad value would otherwise slip through quietly. At every one, invalid input is refused the moment it arrives, with a message a person can act on.

Where the boundaries go

Boundary	What gets checked there	The decision
Configuration	Validated before execution starts, so the running system can trust what it receives.	ADR-001
Workflow structure	A declared graph of steps is validated when it loads, before any step runs.	ADR-002
Stored data	Invariants are enforced where every write path crosses, not only in the interface.	ADR-010
External sources of truth	Tested at the seam where they live, against the real thing rather than a frozen copy.	ADR-004
Generated output	AI output is checked against the rules it cites and the rules pinned for its stack, before anyone sees it.	ADR-012
Irreversible actions	Gated on provable eligibility from every prior answer, never on reaching the last step.	ADR-003
Error messages	Designed like code: lead with the fix, name the likely cause and the next action, and never blame.	ADR-005

Two questions, not one

“Does it run?” is necessary, but it says nothing about the idea. The question that matters is whether it does what the record says the idea must do. A prototype that runs cleanly but misrepresents the idea is a silent failure of intent.

So every must gets at least one check (a test, a fixture or a written walkthrough), and every check is written from the record, never from the code. An AI asked to test its own code will faithfully confirm its own mistakes.

Kind	The question it asks
Faithfulness	Does it do what the record says the idea must do?
Function	Does each feature do what it claims?
Input	What happens with missing, malformed or unexpected data?
State	Can it reach a state the record says is impossible?
Failure	When a step or a dependency fails, does it say so, or carry on quietly?
Human	Can a person understand what happened, and what to do next?
AI-specific	Did the model invent anything, add something unasked, or quietly change a must?

An illustrative example. Imagine a referral tracker for a small charity, meant to flag referrals that go quiet. The prototype runs and its AI-written tests pass, but it counts “overdue” from the day a referral was entered, not the day it was sent. Checks written from the record fail and catch it. Tests written from the code never could.

6 • Decide, then engineer

Continue	Revise	Stop
Every must is met, and the idea still makes sense to the people it is for.	The evidence shows the record was wrong. Change the record first, then the prototype, then run every check again.	The idea is not what it seemed. Record why: a stopped idea with a record is a result, not a waste.

When an idea continues, the record and the checks carry forward, so the real build answers to the same musts. Engineering adds what the prototype left out on purpose: tests in CI on every change, validation at every boundary, error messages written for their reader, observability and documentation.

7 • Fit for purpose: No Silent Failure

Before I call any system fit for purpose, it has to meet six musts. A system missing any one of them is recorded as not fit for purpose, however healthy its dashboards look. The full standard is on my [Philosophy page](#).

- ☐ **Transparency, including the legal limits.** What it accepts, who it covers, what it is legally bound to do and how a decision can be challenged are published early, in language a person can act on.
- ☐ **Failing is allowed, failing quietly is not.** It can conclude it cannot help, as long as it says so, gives the reason and points onward.
- ☐ **Measure what changed for the person.** At least one measure says whether their situation actually changed.
- ☐ **Count the people it turned away.** A system that only reports on its happy path cannot be evaluated.
- ☐ **The system carries the context.** Nobody should need their own paper trail to stop a case falling apart.
- ☐ **Never require what you exist to provide.** A service built to remove a barrier must not make people clear that barrier first.

8 • Checklist

Before prototyping

- ☐ The brief fits on one page, including who is affected without using it
- ☐ Every must is numbered and written so it can be checked
- ☐ States, and the transitions that must be impossible, are listed
- ☐ What must be deterministic is decided, with the reason
- ☐ External contracts come from real documentation or real samples
- ☐ What the prototype leaves out is written down

While building

- ☐ The AI gets the record as its specification in every session
- ☐ One must at a time, one commit each
- ☐ Every assumption the AI lists is checked against the record
- ☐ Nothing goes in that the record does not ask for

Before calling it validated

- ☐ Every must has a check, written from the record rather than the code
- ☐ Invalid input is refused with a message that says what is wrong
- ☐ Nothing fails quietly
- ☐ Someone who did not build it can tell what happened, and what to do next
- ☐ The decision (continue, revise or stop) is written down, with its reason

Need help? For a second pair of eyes on where your system could fail quietly, start a conversation at dobybaxter127.gitlab.io/contact. Reviews are fixed-price and read-only: nothing in your system is touched or changed.

Doby Baxter Computing · reliability engineering, configuration validation, and developer tooling.