Expertly Define

A friendly, practical spec for a tool that keeps product requirements, code, tests, and delivery work connected over time.

Example running through this document: Automation Designer > Version History.

What Expertly Define is

Expertly Define is where you keep the living definition of what your product does. It is not a task board. It is the place you go when you want a clear answer to three questions: What does the product do, where does that live in the code, and how do we know it works?

It stays useful because it follows a few simple rules:

- Every product has a clear tree of requirements, from big areas down to small behaviors.
- Each requirement has a stable key, and its meaning is versioned instead of overwritten.
- Requirements link to the code that implements them and the checks that verify them.
- You can look at the product as-of any release or date, so meaning does not get lost over time.

The voice and tone

Everything in the UI should read like a calm, helpful teammate.

Default sentence style:

> **Users can** …

When you need a system rule:

> **The system** …

For exclusions, the label is:

> **Not included within this requirement**

This keeps requirements crisp, durable, and easy to scan.

What problems it solves

- Teams forget what parts of the product were meant to do, especially after a year or two.
- Changes feel risky because it is hard to see what a change might affect.
- Teams have tests, but cannot easily tell which tests protect which product behaviors.
- Tickets get closed and disappear into history, so they do not preserve product meaning.

Expertly Define fixes this by keeping meaning, implementation, and verification connected.

How requirements are structured

A product is represented as a tree. Every node in the tree is a requirement at some level. You do not need different object types for modules versus features versus sub-features. It is just levels of zoom.

Key behaviors:

- A requirement can be high-level or very specific.
- You can move items in the tree without breaking their identity.
- You can tag requirements as functional, nonfunctional, security, performance, usability, or invariant rules.

Stable keys and version history

Each requirement has a stable key that never changes. That key can be used in code and tests. When the meaning changes, you create a new version. You never overwrite the past.

- If something is no longer needed, you mark it canceled. It stays findable.
- If something is replaced, you mark it superseded and point to the replacement.
- You can compare versions and understand what changed and why.

Linking requirements to code

Each requirement keeps a list of where it lives in the code. This is the core safety feature. When you need to change behavior, you can jump straight to the right places instead of guessing.

- Links can be created by an IDE plugin or by a mapping file, depending on your preferences.
- The system rescans after merges and marks links that need reconfirmation when code moves.
- If you want, the system can open a mapping pull request that only updates requirement markers or mapping metadata.

Linking requirements to checks and tests

A requirement is only truly safe when you have evidence that it works. Expertly Define lets you link tests and other checks directly to requirements.

- Verification can include unit, integration, end-to-end tests, manual checks, and monitoring signals.
- The UI should show a simple status: not verified, verified, or failing.
- When a check starts failing, the requirement should show that clearly.

Keeping the system honest

Traceability systems die when links go stale. Expertly Define treats this as normal and manageable. It provides a tidy 'fix-it list' for broken links, missing checks, and orphan markers.

- Requirements with no code links are flagged.
- Requirements with no checks are flagged, especially if they are tagged critical.
- Code markers that reference missing or canceled requirements are flagged.
- Recent meaning changes trigger a short impact checklist so children and dependents stay consistent.

Import and discovery

To avoid a slow start, Expertly Define supports bootstrapping from what you already have. Everything starts as a draft and becomes official only when a human accepts it.

- Import from documents: upload an FSD or notes and get a draft tree.
- Import from code: scan repos, draft a starting tree, and propose code and test links.
- Import from a test site: explore staging, take screenshots, infer current behavior, and flag contradictions.

After any import, the system creates a short questions inbox. You answer by typing or speaking, preview the changes, and then apply them.

AI features that help without being annoying

- Draft subtree from typed or spoken notes, with a preview panel and easy accept or reject.
- Suggest acceptance criteria and test ideas, including edge cases and negative cases.
- Suggest likely code locations for a requirement, then open a mapping PR after you approve.
- Detect possible contradictions between requirement text, code behavior, and checks, and turn them into questions.

Jira and Teamwork integration

Jira and Teamwork remain the place where work is assigned and scheduled. Expertly Define stays focused on meaning. The bridge is simple: linked delivery work.

- Each requirement can show related delivery work, including intent like implements, modifies, verifies, or refactors.
- Once a requirement or subtree is approved, Expertly Define can draft delivery stories as a head start.
- Stories should be pushed as drafts so a PO can edit and approve before they become real work.

Exports and 'book' PDFs

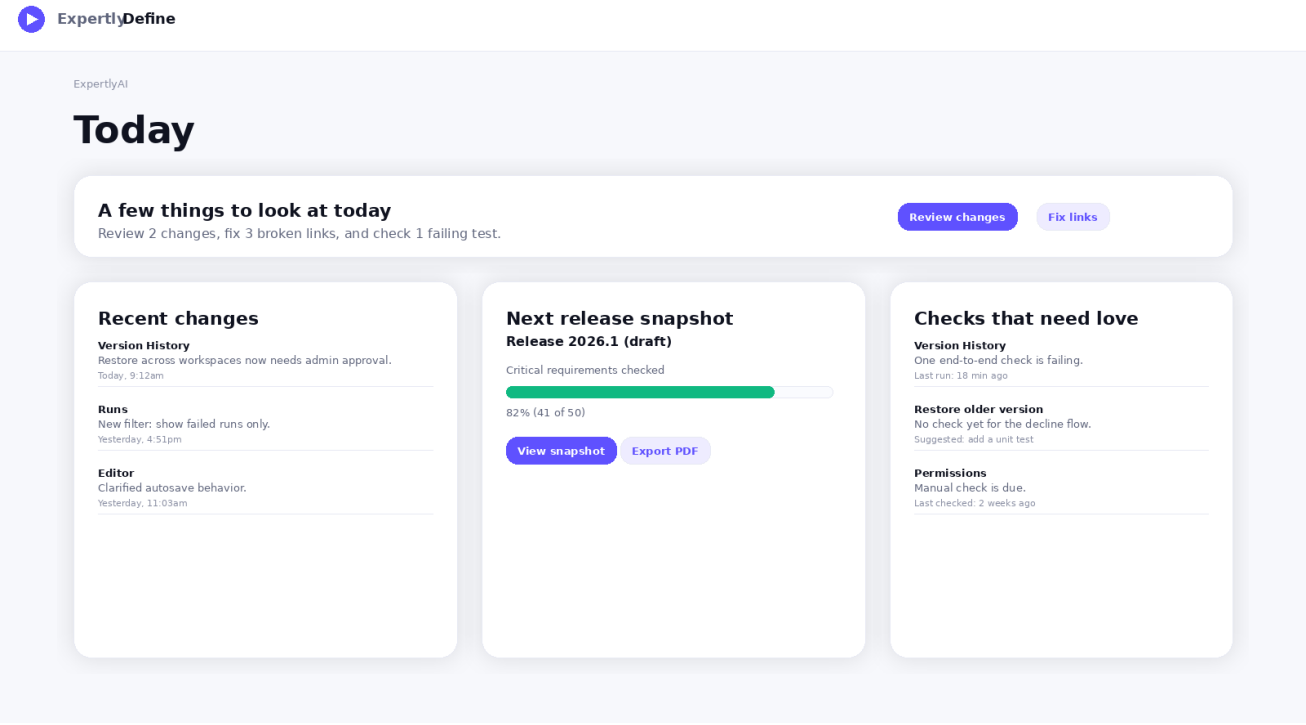
- Product book: a skim-friendly tree with one or two lines per item.
- Requirements book: detailed pages per requirement using the template you defined.
- Release snapshot pack: what changed, what is checked, and what still needs attention.

Exports should be clean and pleasant to read, because they help teams align quickly.

End-to-end example workflow

Here is the end-to-end flow for Automation Designer > Version History. A meaning change adds admin approval for cross-workspace restore. The team reviews impact, drafts Jira stories, confirms code links, fixes a failing check, creates a release snapshot, and exports a requirements book.

01 Dashboard



03 Requirement Definition

ExpertlyDefine

ExpertlyAI

Automation Designer > Version History

Definition | Versions | Implementation | Verification | Delivery work

What this does
Users can view, compare, and restore previous versions of an automation.

Why this exists
This helps people understand changes over time and recover safely from mistakes.

Not included within this requirement
- Branching or merging versions
- Restoring only part of an automation

How we know it works
- Users can see a list of versions with author and timestamp.
- Users can compare any two versions and clearly see what changed.
- Restoring creates a new version and does not overwrite history.
- Only admins can restore across workspaces.

At a glance
Ready to build
Implemented
1 check failing

Linked implementation
4 locations

Linked verification
7 checks

Quick actions
Edit requirement
Export PDF book

05 Change Review

ExpertlyDefine

Automation Designer > Version History

Review changes

What changed (v3 to v4)

Added	Restoring across workspaces now needs admin approval.
Added	A restore request can be declined, and the requester is notified.
Clarified	Restoring always creates a new version, even if identical.

Why
Support asked for safer restores after a few accidental rollbacks between workspaces.

What to review
This is the short checklist for a safe update.

- **Implementation: restore_service.py**
  Add admin approval check
- **Implementation: RestoreDialog.tsx**
  Show approve/deny UI
- **Verification: e2e restore check**
  Fix failing approve/deny path
- **Verification: unit permissions test**
  Add decline edge case
- **Child requirement: Permissions**
  Confirm it still matches

Approve and make current
Create Jira follow-ups

06 Story Drafts

ExpertlyDefine

Automation Designer > Version History

Draft Jira stories

Proposed drafts

Push drafts to Jira

These are drafts. A PO can edit them before they hit the real backlog.

Story 1: Version timeline UI
Users can see versions with author, time, and notes.
Acceptance: list loads fast for 200 versions.
Edit

Story 2: Compare versions
Users can pick any two versions and see a clear diff.
Acceptance: show changes only toggle.
Edit

Story 3: Restore + audit
Restore creates a new version; log who restored and why.
Acceptance: restore is permission-checked.
Edit

Story 4: Admin approval
Cross-workspace restore requires admin approval.
Acceptance: requester is notified on approve/deny.
Edit

07 Implementation Links

ExpertlyDefine

Automation Designer > Version History

Implementation

Where it lives in code

Rescan now | Open mapping PR

Links refresh after merges. If something moved, we will ask you to confirm it.

Status	Path	What it does	Last checked
Up to date	automation/versioning/history_service.py	Loads the timeline	Today
Up to date	automation/versioning/restore_service.py	Restores an older version	Today
Needs a look	ui/version_history/RestoreDialog.tsx	Restore UI and admin approval	After last merge
Up to date	api/routes/version_history.py	API endpoints	May 19

10 Export Preview

ExpertlyDefine

Automation Designer > Version History

Export

Requirements book

Choose what to include, then export a clean PDF you can share.

Include

- [x] Tree (always)
- [x] What this does and why this exists
- [x] Not included within this requirement
- [x] How we know it works
- [ ] Related delivery work
- [ ] Implementation links
- [ ] Verification links

Export PDF

Preview

Automation Designer

Version History

Users can view, compare, and restore previous versions of an automation.

UI field labels (final)

This is the on-screen template for any requirement.

Section	What goes here
What this does	One clear sentence that starts with “Users can …”
Why this exists	One or two sentences in plain English
Not included within this requirement	Bullets that avoid confusion and scope creep
How we know it works	Bullets that read like checks and can map to tests
Related delivery work	Jira or Teamwork links with intent labels
Implementation	Code locations with last confirmed info
Verification	Tests and checks with current status