The Low Code Factory

ENGINEERING WHITE PAPER

Compiling legacy forms into React.

A two-stage compiler that captures every construct in a vendor form definition, emits code a developer would have written, and reports precisely what it could not translate.

THELOWCODEFACTORY.COM HELLO@THELOWCODEFACTORY.COM JULY 2026 · V2.0

Abstract

A manufacturer ran hundreds of business-critical forms on an aging, proprietary ERP forms platform: leave requests, security entries, purchase approvals, plant operations. The forms worked, which was precisely the difficulty. They ran the business, they were tied to a vendor runtime the company no longer wanted to license, and the people who understood the platform had moved on.

Rebuilding them by hand was estimated by the customer's own developers at two days per form. Across the estate that is a headcount decision rather than a project, and it produces no new functionality — a business case no sponsor approves. Which is why estates like this one remain on a platform long after everybody agrees they should not.

This paper describes the engine built instead: a compiler with two intermediate representations, in which a lossless Source IR keyed on the *source* vocabulary makes silent loss structurally impossible, and a per-form migration report accounts for every construct — translated, or explicitly preserved for a developer to complete.

187
forms migrated, plus a dashboard portal

4×
faster than the hand-migration estimate

100%
of every form captured; nothing silently dropped

SECTION 01

What a legacy form actually is

A form on a platform of this kind is not a layout. It is a declarative specification spanning four planes, and a faithful modernization has to preserve all four.

PLANE	CONTAINS
Models	Data bindings and their types
View	Controls, grids, layout
Behaviour	Event and field scripts that run as the user interacts
Actions	The lifecycle that fires on load, save and submit

Multiply that by several hundred screens and the problem stops being one of effort. It becomes one of **consistency**, because no two developers reading the same legacy form will interpret it identically. Fields drift, validation is missed, and the inconsistency surfaces only in testing — or in production.

SECTION 02

Why the obvious approaches fail

- **Hand-rewriting.** Two days per form, several hundred forms, no new functionality at the end. It is not merely slow; it is inconsistent, and the inconsistency is invisible until late.

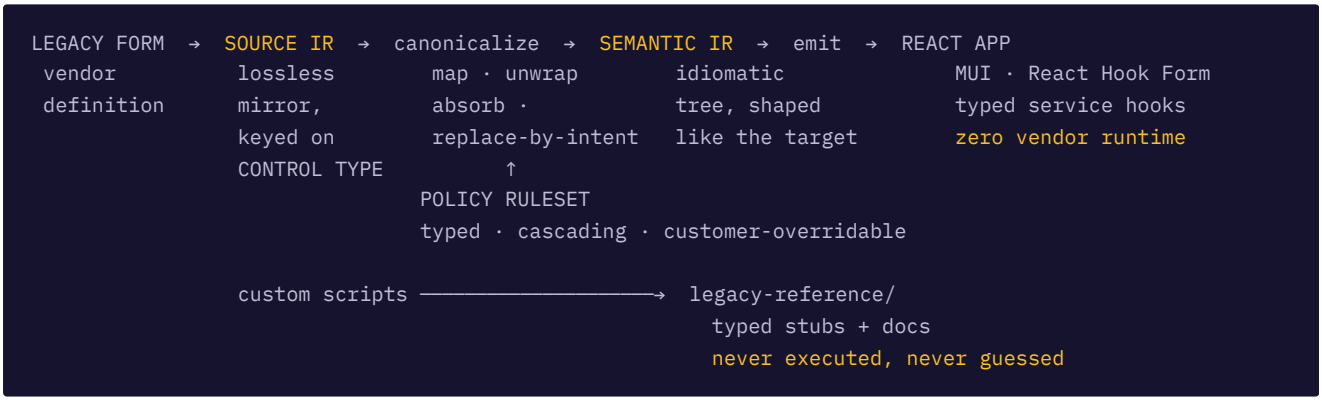
- **Wrapping the old runtime.** An iframe around the legacy platform preserves the licence bill, the lock-in and the dated interface. It relocates the problem rather than solving it.
- **Naive conversion.** A tool that translates what it recognises and skips what it does not will produce something that renders. The gap surfaces in user acceptance testing, and by then the entire output is suspect.
- **Black-box generation.** Output no developer can read or extend is simply a new kind of lock-in: a runtime the customer could not change, swapped for a codebase they cannot either.

> THE BAR THIS SET
>
> The output must be code a developer would have written by hand. Every construct that cannot be auto-translated must be **visibly flagged, never dropped**. And no vendor runtime may survive into production.

SECTION 03

Architecture: a two-stage compiler

The engine does not translate a form line by line. It parses it, understands it, and re-expresses it. That is the difference between a converter and a compiler, and it is the reason the output is readable.



The Source IR, and why it is keyed on the source

The first stage produces a **Source IR**: a lossless mirror of the legacy form, keyed on the *original control type*.

This is the single decisive design decision in the engine, and it is a small one with a large consequence. Because the intermediate representation is keyed on the source vocabulary rather than the target's, the engine **physically cannot drop a construct it does not yet understand**. Anything unrecognised is still present in the IR, still counted, and still reported.

Every converter that quietly discards what it does not recognise made the opposite choice — usually without noticing, because keying on the target vocabulary is the natural thing to do and the loss is invisible until somebody looks for it.

Canonicalization to the Semantic IR

Canonicalization passes then fold the Source IR into a **Semantic IR**: an idiomatic tree shaped like the target rather than the source. The passes are named for what they do — mapping one construct onto another, unwrapping needless nesting, absorbing a wrapper into the thing it wraps, and replacing a

legacy idiom with the modern construct that expresses the same intent.

All of it is governed by a typed, cascading **policy ruleset** that the customer can override. A quirk specific to one estate is handled by adding a rule, not by forking the engine — which is what stops a second customer from becoming a second product.

Deterministic emission

React is generated from the Semantic IR, and the transform is deterministic: the same form in produces the same React out, on every run.

That property does more work than it appears to. It means the migration can be re-run as the source estate evolves, rather than being a one-way door. It means a reviewer can diff two runs instead of re-reading the whole output. And it means a change to the policy ruleset shows its effect as a diff across every affected form at once, rather than requiring a manual audit.

SECTION 04

The line between structure and logic

Most of a form is structural: its fields, its validation, its grids, its layout. Structure can be translated deterministically, and the engine emits it as native React using MUI, React Hook Form and typed service hooks over TanStack Query. It reads like code a senior engineer wrote by hand, because that was the standard it had to meet.

The custom business logic is the genuinely hard part, and it is the one place where silent automation is dangerous. A machine translation of a business rule that is subtly wrong is worse than no translation at all, because nobody will look for the error.

So the engine draws a hard line and **never executes a custom script**. Each one becomes a typed stub handler in the generated component, plus read-only reference documentation placed alongside the form it belongs to, in `legacy-reference/`. A developer completes it by hand with the original in front of them and the context intact.

> WHY THE LIMIT IS WHAT MAKES IT TRUSTWORTHY
>
> The engine could have attempted the custom logic and produced something plausible. It would have been wrong occasionally, in ways nobody could easily detect, and the entire output would then have been suspect. Capturing everything and being precise about which part still needs a person is what makes the number defensible.

SECTION 05

The migration report

Every construct in a source form is accounted for. It is either translated into React, or explicitly recorded as preserved for a developer to complete — and the report states which, per construct, per form. The count is exact.

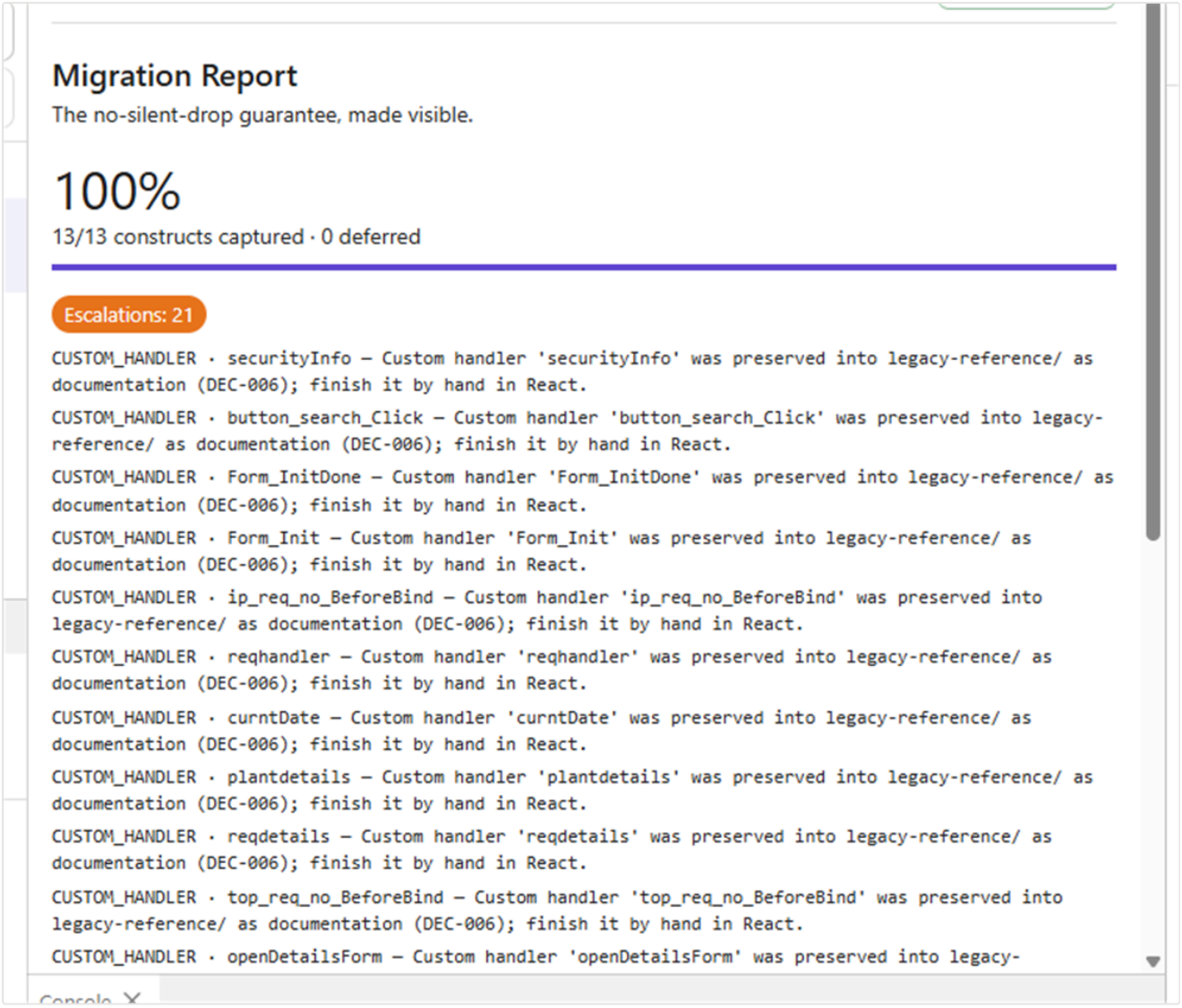


Figure 1. The migration report for one form: every construct captured, none deferred, and a per-construct log of exactly what became React and what was preserved as a documented handler. This is the artifact that makes the programme governable — a number a programme owner can take to a steering committee before cutover.

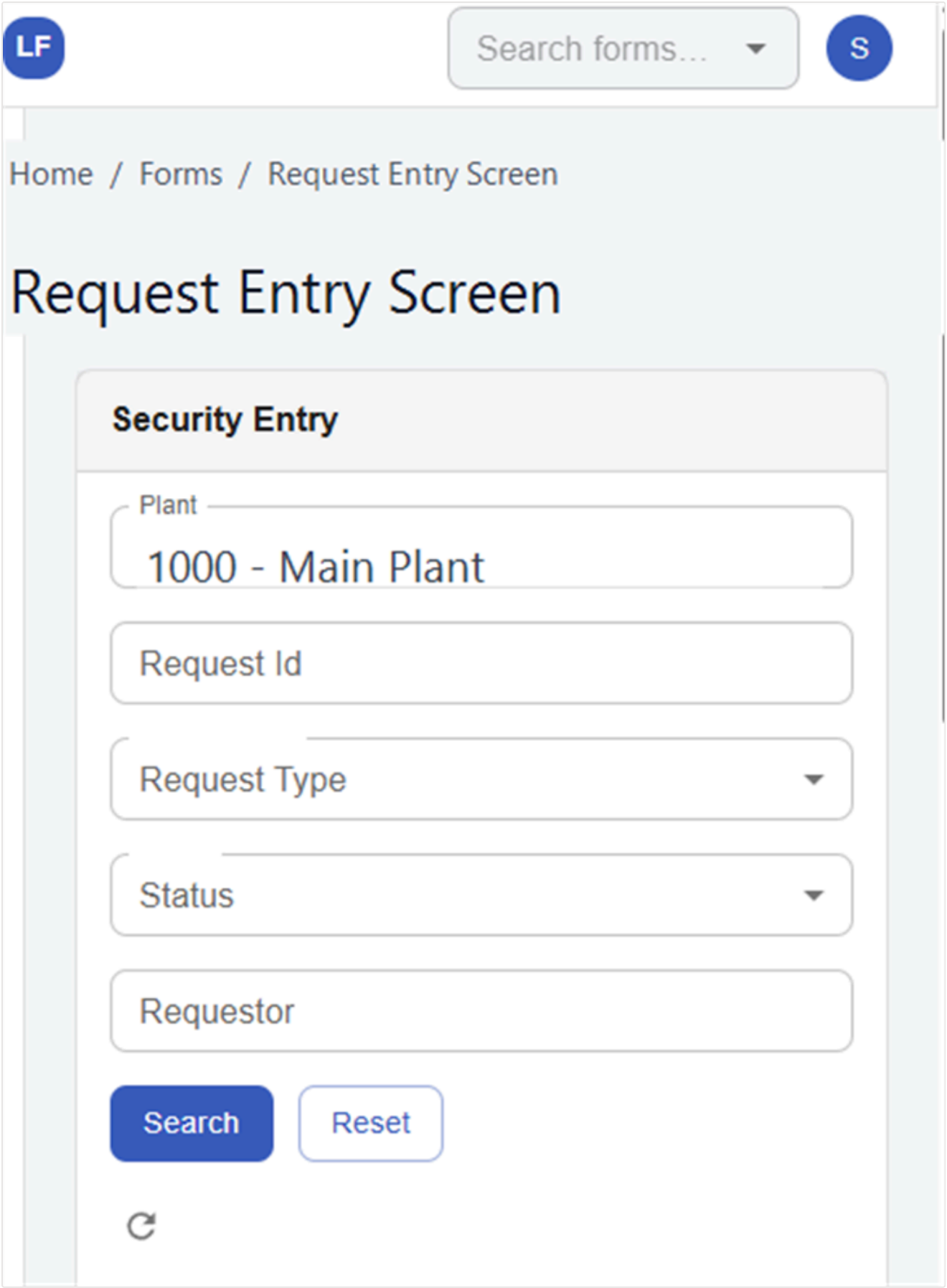


Figure 2. A generated React screen, rendered from a migrated form definition. Fields, a Material-styled layout, validation and a bound results grid — the structural majority a developer would otherwise have hand-written, produced deterministically and identically on every run. Sample form; no customer data is shown.

SECTION 06

Delivery: a tool they operate, not a service we perform

The engine could have been run by us, form by form, with the output delivered as a series of pull requests. That would have been faster to start, and it would have left the customer dependent on us for every subsequent form, every ruleset change and every correction as the estate continued to evolve.

Instead the engine ships as a tool their developers operate. A form is selected, compiled, and its migration report reviewed. The generated React is inspected in a live preview beside the original. The policy ruleset is theirs to extend as the estate reveals new quirks, and it is versioned and exportable rather than embedded in the engine.

The remaining custom logic is completed by a developer against the reference documentation the engine produced. That work is real, it is skilled, and it is deliberately left to a person.

SECTION 07

Results

187 forms and a dashboard portal have been migrated for the first customer.

Hand-migration had been estimated by their own developers at two days per form. With the engine, a developer completes **two forms per day**: the structural majority arrives generated, and the remaining custom logic is finished by hand in roughly half a day with the reference documentation alongside it.

	BY HAND	WITH THE ENGINE
Per form	2 days	0.5 days
187 forms	~370 developer-days	~95 developer-days
Consistency	Varies by developer	Deterministic; identical on every run

The economic point is the one that mattered. The engineering here did not make a possible project faster. It made an *impossible* project possible — turning a business case that no sponsor would approve into one that was straightforward. That is usually where the value of building a tool lies, and it is why building the compiler was cheaper than doing the work.

The generated application depends on React, MUI and the customer's own conventions, with no vendor frontend library anywhere in the output and no runtime shim underneath it. The customer owns the engine, the policy ruleset, the generated code, and the route off the platform they were leaving.

SECTION 08

Limitations

- **Custom business logic is not translated.** By design. Roughly the last fifth of each form is completed by a developer. The engine tells you exactly which fifth; it does not do it for you.

- **The policy ruleset requires curation.** A new estate will surface constructs the rules do not yet cover. They are reported rather than dropped, but somebody has to write the rule.
- **Output fidelity is to the specification, not to the pixel.** The generated screens are idiomatic React using the customer's conventions. They are not a pixel-for-pixel reproduction of the vendor's rendering, and reproducing one was never the goal.
- **Deterministic means deterministic.** Hand-edits to generated files will be overwritten by a re-run. Completed logic lives in the stub handlers and in `legacy-reference/`, which are preserved.

SECTION 09

Conclusion

The decisive design choice was keying the first intermediate representation on the source vocabulary rather than the target's. It makes silent loss structurally impossible, because a construct the engine does not understand still has somewhere to live and still appears in the count.

The second was admitting the limit. A tool claiming fully automated conversion of custom business logic is either wrong or producing code somebody will quietly rewrite later. Drawing a hard line, and reporting exactly what fell on the other side of it, is why a programme owner can take the number to a steering committee — and why the number is worth taking.

The Low Code Factory

An enterprise solutions company. We design, build and modernize business-critical applications, workflows, integrations and internal platforms.

thelowcodefactory.com · hello@thelowcodefactory.com · linkedin.com/company/tlcfworks