Two people. Five months. A platform the market didn’t have yet.

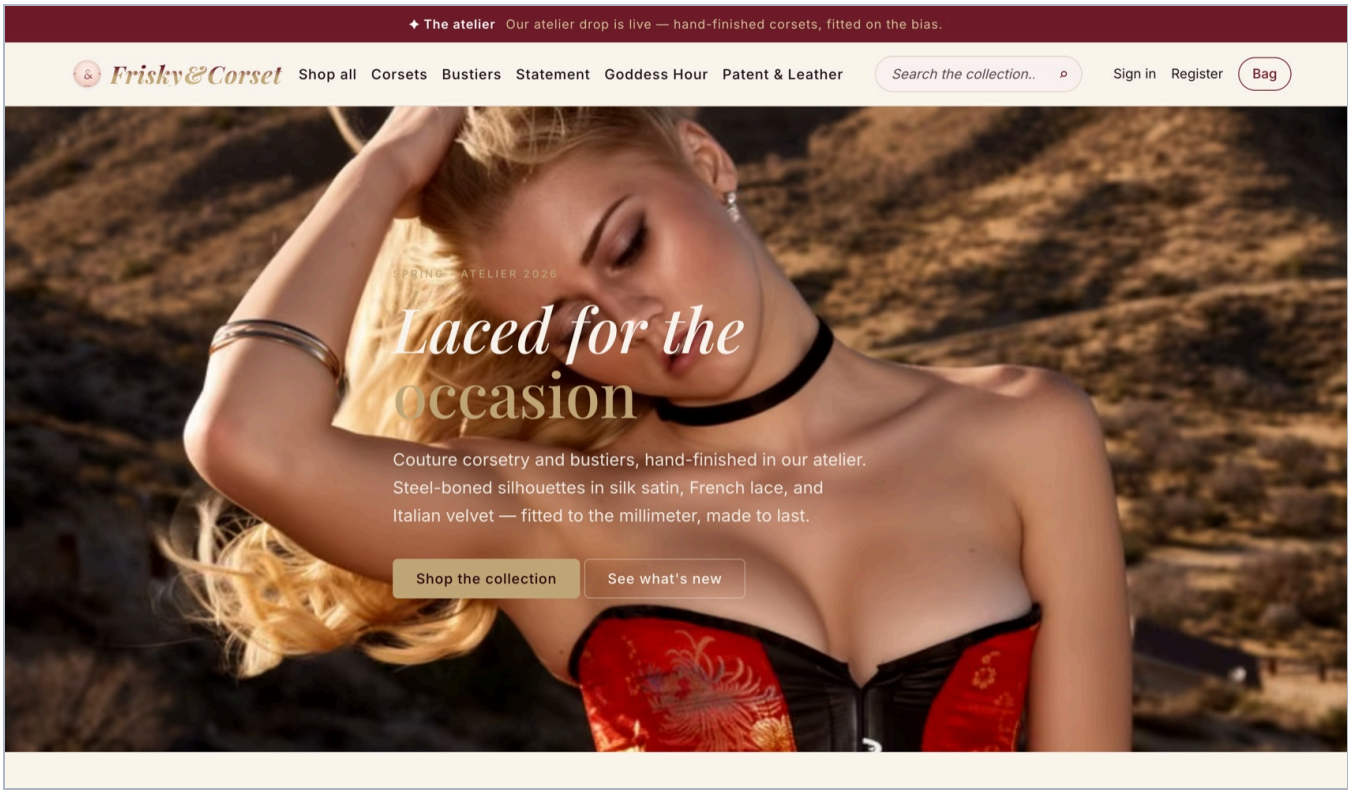
A ProjectThunder case study in building custom software with AI — including the part where it found a real bug in our own code.

The short version

In May 2026 we started rebuilding a twenty-year-old commerce system. By October we had a production-grade platform: **137,000 lines of first-party code, 1,240 automated tests, 200 routes, 122 data entities** — and an interface for AI agents that, at the time we shipped it, no commercial platform offered.

Two people built it. One of them part-time.

That is the kind of scope normally quoted at eighteen months and a team of eight. We want to show you exactly how it was done, because the method is what we sell — not the product.



The storefront, running from a clean install. Server-rendered, so the price in the page is the price — which matters when the visitor isn’t a browser.

The situation

ProjectThunder had been running commerce software since 2004. The existing system worked. It also couldn't take the next step: AI agents were beginning to shop and transact on behalf of real buyers, and nothing in the system — or in any platform we could buy — was built to let a machine do business safely.

The conventional options were both bad. Bolt an API onto a twenty-year-old codebase and inherit every assumption it made about a human being on the other end. Or start clean and spend two years rediscovering business rules that were already encoded, correctly, in software that had been taking orders for two decades.

The decision that shaped everything

We used the old system as a **specification, not a foundation.**

All 112,973 lines of it were read, mined for rules, and left behind. Not forked, not strangled, not incrementally migrated — read. When we later measured what survived of the original scaffolding in the new codebase, the answer was **511 lines**.

That sounds reckless. It was the opposite. Twenty years of production code is the most accurate requirements document a business will ever own — every edge case someone phoned in about is in there somewhere. What it is *not* is a good place to stand while you build something with a different shape. Treating it as a spec let us keep the hard-won knowledge and discard the architecture that no longer fit.

If you have a system like this, that judgment call is probably the most valuable thing in this document. Most vendors will quote you an incremental migration because it feels safer and bills longer. Sometimes it is right. Often it is how a two-year project becomes a four-year one.

How we actually build

Here are two numbers. Together they are the whole method.

83.8%	**5%**
of the surviving first-party code was written in AI-assisted commits.	of the findings from our automated code review turned out to be real.

The first number is leverage. The second is why the first one is safe.

AI writes most of the lines here, and we are matter-of-fact about that. It does not write the decisions. Every architectural call, every trade-off, every "no, that's wrong" came from a person — and the verification layer exists precisely because generated output cannot be trusted on sight.

That second number deserves unpacking, because it is the one most AI-development stories leave out. We ran a multi-reviewer pass over a large change set: **twenty reviewers produced 149 findings. Eight were real.** The other 141 were noise — plausible, well-argued, confidently wrong.

If we had merged on the strength of "the AI reviewed it," we would have shipped a mess and chased phantoms. The work is not generating findings. The work is **verification**, and it is stubbornly human.

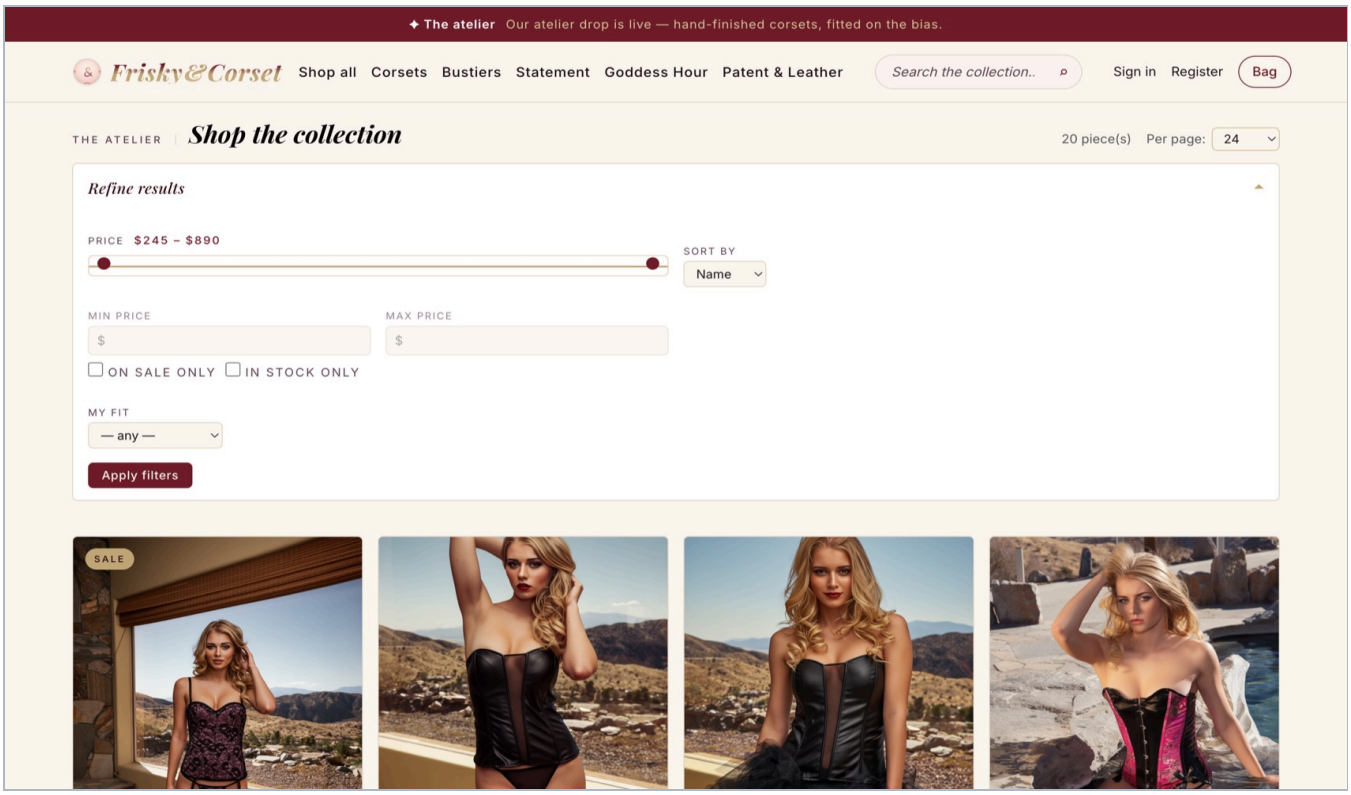
One of those eight was an account-takeover vector in our own password-reset flow. We found it because we look. It is in this document because a case study that only lists wins is marketing, and you should not buy engineering from someone who shows you only wins.

What quality control looks like here

- **1,240 automated tests**, zero skipped, full suite in 57 seconds. Fast tests get run; slow tests get ignored.
- **Roughly 2:1 production-to-test code.** Tests are treated as deliverables, not afterthoughts.
- **A browser-driven end-to-end suite** that exercises the real application, not mocks.
- **Dual continuous integration** — GitHub Actions and an on-premises Azure DevOps Server — so the build is verified in two independent environments.
- **A database-drift gate.** Every build checks that the data model and its migration scripts still agree across all three databases. It is impossible to quietly change a schema and forget the migration. This one is unusual, and it is the kind of thing that prevents a 3 a.m. deployment failure.

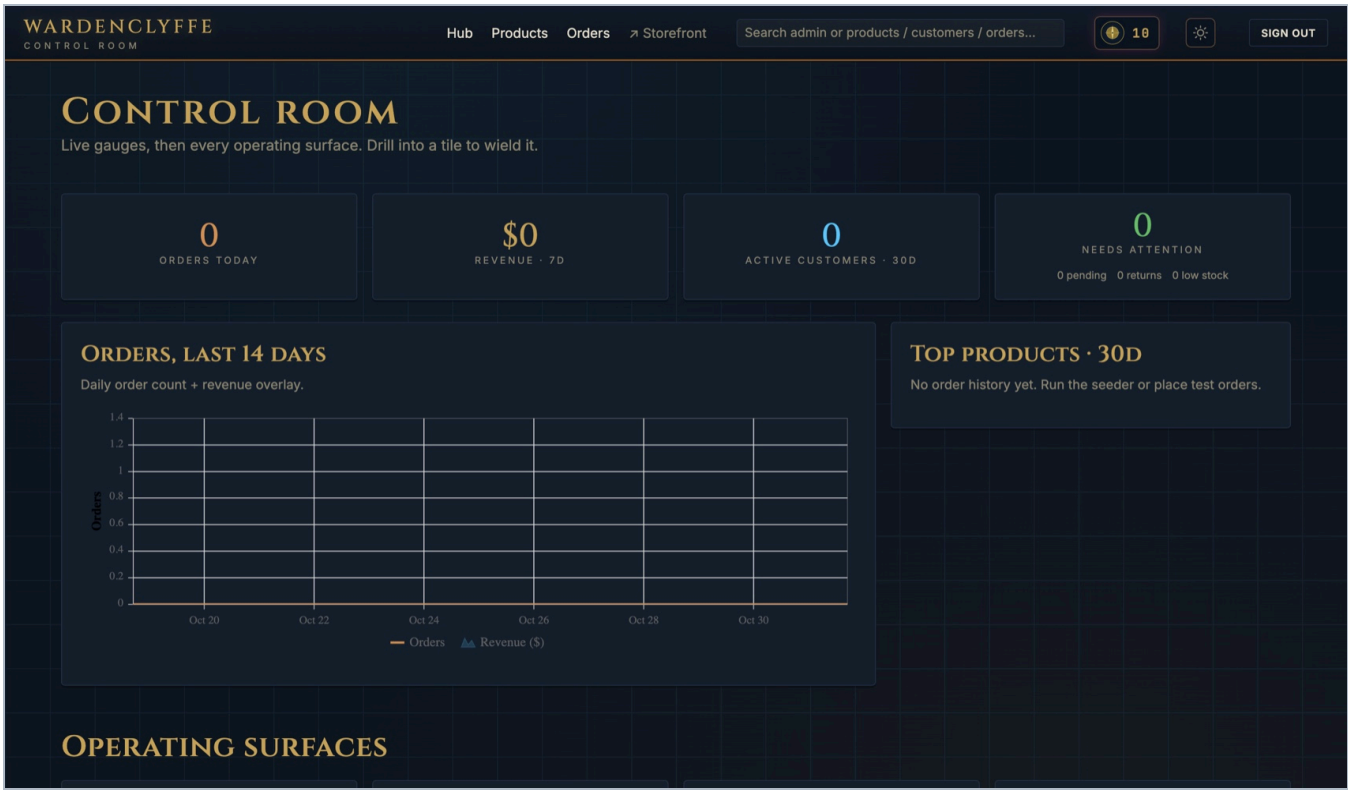
What exists now

First-party code	**137,390 lines**
Production C#	**95,394 lines**
Automated tests	**1,240 passing**
Routes	**200**
Data entities	**122**
Database migrations	**38**
Agent tools	**50**
Elapsed time	**151 days**
People	**2**

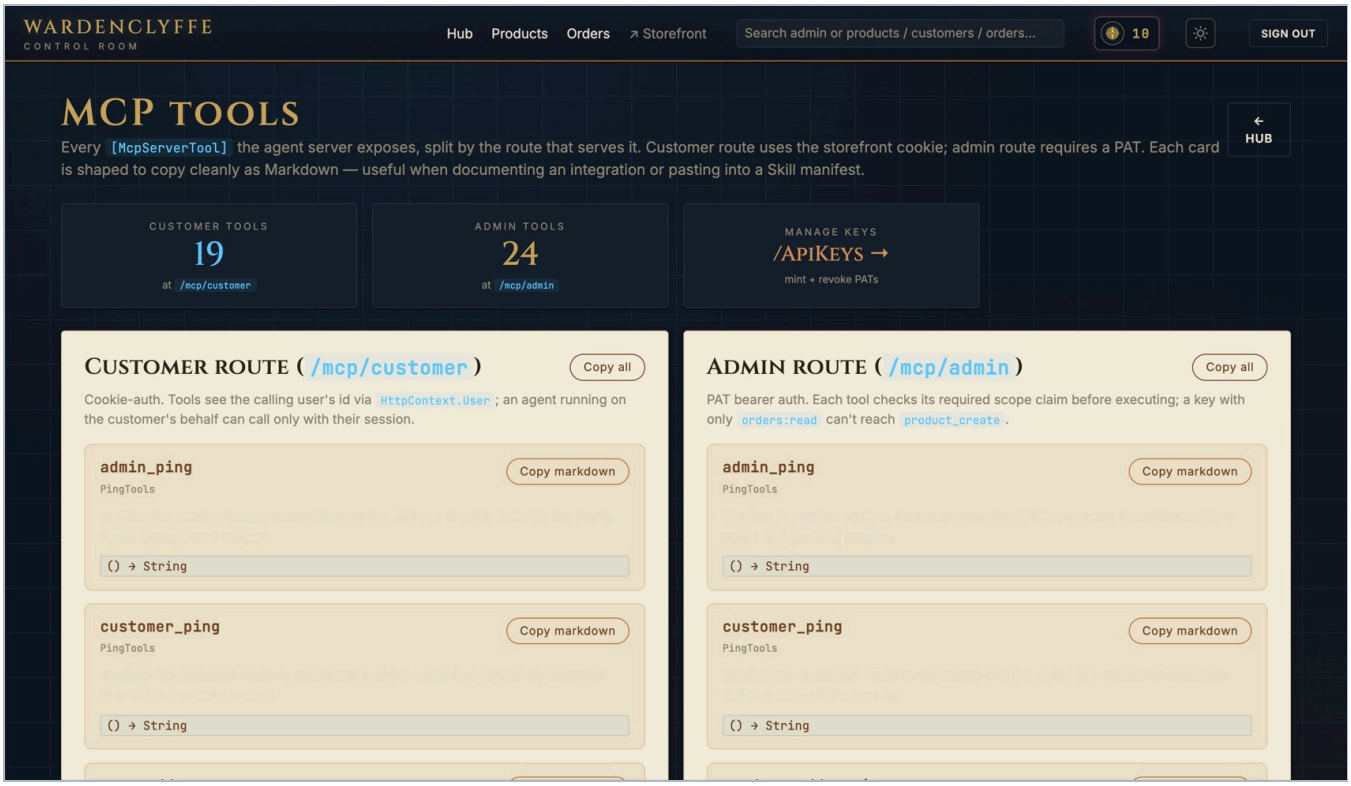


Faceted browse. Price, availability and a saved-profile fit filter — the same narrowing machinery works for vehicle compatibility, entitlement tier or clinical eligibility.

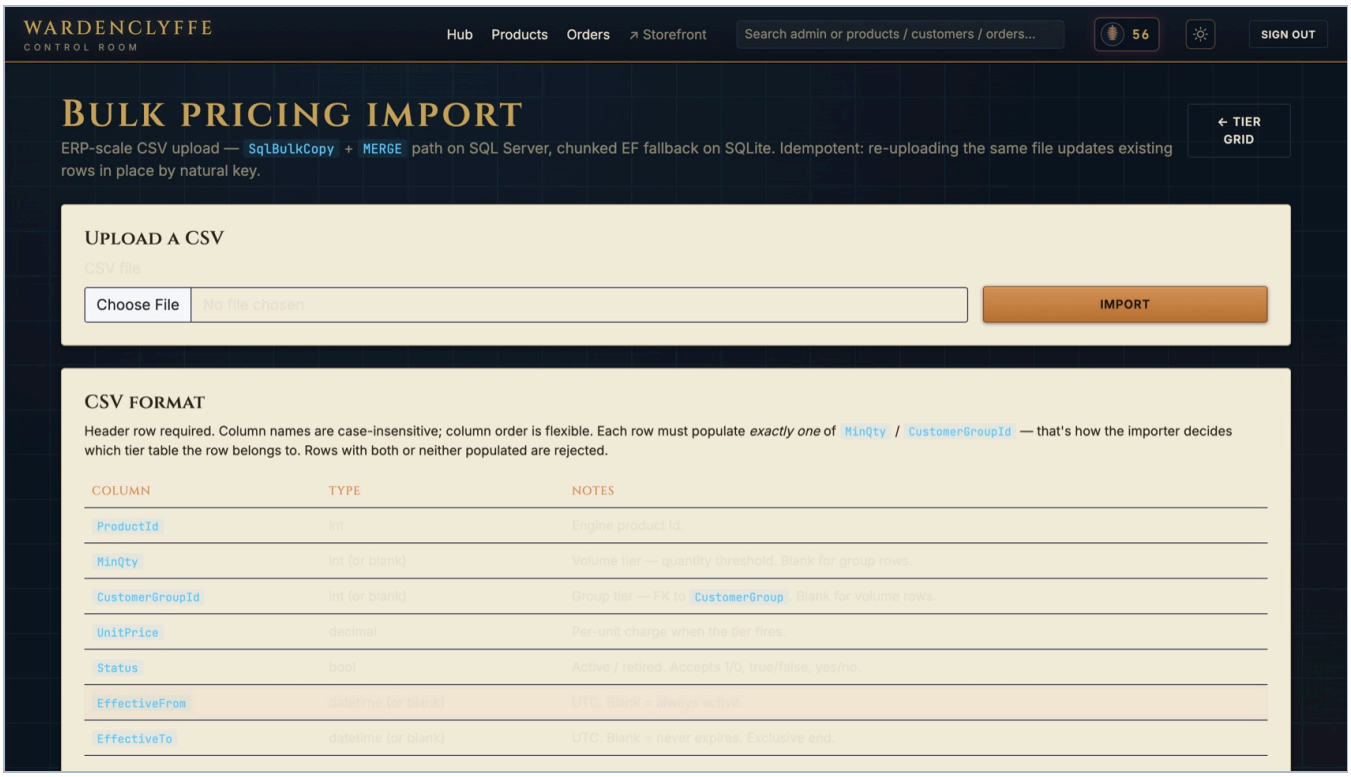
Storefront, administration, catalogue, pricing tiers, bulk import, returns, audit logging, search — plus a complete interface that lets an AI agent browse, add to cart, and check out through the same permission rules a human faces. One set of rules, enforced once, whether the customer is a person or a program.



The operator’s control room. Ninety-nine administrative screens across catalogue, operations, marketing and system.



The agent interface documents itself: a customer route and an administrative route, each tool gated by the caller's own permissions — the same rules the human screens enforce.



Bulk pricing import. Built for the systems that already feed your catalogue: compressed uploads to a 200 MB window, streamed straight past the ORM, and safe to replay.

It is source-available under the Business Source License 1.1, free to run in production under $1M in annual sales, and converts to Apache 2.0 in May 2029.

What this means for your project

You are probably not building a commerce platform. The transferable parts:

We can start from your existing system without being trapped by it. We read legacy code as specification. Most of what your old system knows is worth keeping; most of how it is built usually is not.

Correctness under failure is designed in, not patched on. Payment writes survive retries without double-charging. Every privileged action lands in an audit log. Those aren't commerce features — they are what any system handling money, records, or automated actions needs.

Speed does not come from skipping the tests. It comes from AI doing the typing while people do the deciding, with a verification layer sized for the fact that most generated output is wrong.

You will see the evidence. We built this one for ourselves, which means we can open the repository, the commit history, the test results and the review findings to you in a way client work under NDA never allows. Ask us for the measurement appendix — it is twenty-four pages and every number traces to a command you can re-run.

ProjectThunder.com, Inc. — building commerce and custom software since 2004.

Tell us where you are, where you want to be, and what you're trying to protect along the way.

Start a conversation →