

TECH & AI CASE STUDY - PROPTECH - 2026

Cloud-Native Rebuild That Tripled Feature Velocity for Proptech

Based on a delivery engagement with Fairbrick Living

3x

FEATURE VELOCITY

**3 months to <2
weeks**

LEAD TIME TO PRODUCTION

-70%

PORTAL PAGE LOADS

8 to 30+

PRODUCTION CHANGES/MONTH

A fillable practice edition: the full story, 8 actions,
6 reflections and a 90-day plan - work it with your team.

Free - no email required
emersontechnology.com

MAKE THIS PRACTICE BOOK YOURS

Work it, do not just read it

This practice book turns the Fairbrick Living case study into a working session for your team. Read the full story first - what happened, what was built and what it changed. Then tick the 8 actions you can commit to, write short answers to the 6 reflections, and finish with a 90-day plan. Every field in this PDF is fillable: open it in Adobe Acrobat Reader, Chrome, Edge or macOS Preview, type directly into the boxes and save.

YOUR NAME	ROLE
COMPANY	TEAM / DEPARTMENT
DATE	EMAIL (OPTIONAL)

MY GOAL FOR THIS PRACTICE SESSION

- 01 The story** - What happened at Fairbrick Living - before, during and after the project.
- 02 The results** - The numbers that changed, plus industry benchmarks from public research.
- 03 The 8-action checklist** - Tick boxes as your organization completes each step. Owner and date fields included.
- 04 The 6 reflections** - Fillable questions that pressure-test your own roadmap.
- 05 My 90-day plan** - Three fillable phase cards that turn this case study into your roadmap.

RUN IT AS A 90-MINUTE TEAM SESSION

25 min READ Read the story together. Ask everyone to note the one number that surprised them most.	40 min PRACTICE Work the 8-action checklist and the 6 reflections. Assign an owner and a date to every action you commit to.	25 min COMMIT Fill in the 90-day plan. One goal per phase, one number that tells you it worked. Book a review date before you leave the room.
--	--	---

THE STORY

What happened, and why it worked

01 The Company

Fairbrick Living is a proptech company that builds the operating layer for residential property management: tenant and owner portals, rent collection, leasing workflows and maintenance dispatch. The platform serves property managers overseeing 12,000 rental units, and its product sits in the unglamorous but mission-critical middle of the rental economy — when a water heater fails or a rent run processes on the first of the month, property managers expect Fairbrick's software to simply work. The company employs a mid-sized product and engineering organization that ships continuously to tenants, owners and property managers through web and mobile interfaces.

That market context matters for one specific reason: speed is the product. Property managers choose between Fairbrick and competitors on how quickly new workflows arrive — new payment options, new inspection checklists, new owner reporting — because those features are what they resell to their own boards and owners. McKinsey research reports that companies in the top quartile of developer velocity achieve revenue growth up to five times faster than bottom-quartile peers, and Fairbrick's leadership had read that research with growing discomfort, because their own feature velocity had been slowing for years while their roadmap backlog grew.

02 Where Things Stood

Fairbrick's entire platform ran on a 10-year-old PHP monolith with a jQuery front end, nightly cron jobs and hosting on a single aging VPS. The system had accreted a decade of features into a codebase where modules reached into each other's database tables, and nobody could change one workflow without tracing side effects through three others. Feature lead times stretched past three months, and roughly 60% of engineering capacity went to untangling side effects rather than building new capabilities. There was no automated test suite, so manual QA caught what it could — and the rest surfaced in production, often at the worst possible moments.

Some failures were so regular they had become part of the company's folklore. Rent-collection days — the first of every month, when payment volume spikes — brought predictable outages, and the operations team planned around them the way farmers plan around weather. Mobile users abandoned application flows at twice the desktop rate because the jQuery front end had never been designed for small screens or weak connections. The situation turned acute when a security disclosure on the unsupported framework version forced an emergency patch weekend, consuming the entire engineering team's attention while feature work stopped. Around the same time, two enterprise property-management prospects lost RFP competitions and cited the platform's sluggishness in their feedback, and recruiting had become its own crisis: engineers who interviewed with the team asked pointed questions about the stack and accepted offers elsewhere.

03 The Challenge

The challenge was a platform whose cost of ownership had quietly become the company's largest strategic liability. Stripe's Developer Coefficient survey reportedly found developers spending a third or more of their time on technical debt and maintenance rather than new features — Fairbrick's engineers were spending closer to 60%, which meant the company was paying full engineering salaries for what was, in effect, archaeology. DORA research consistently links stronger software delivery performance to better overall organizational outcomes including profitability and market share, and Fairbrick sat at the wrong end of that correlation: deployments were rare, risky events that everyone feared.

The rewrite question carried well-known dangers, and the team was honest about them. A big-bang rewrite would freeze feature work for quarters while competitors shipped, and industry folklore is full of rewrites that never

finished. Continuing on the monolith was equally untenable: the security disclosure proved the framework was a liability, the lost RFPs proved customers noticed, and the recruiting problem proved the talent market had voted. The real challenge was therefore a third path — rebuilding the platform module by module while a live business kept running on the old one, with the property managers' rent runs, maintenance tickets and leasing pipelines never interrupted. It also required an organizational commitment: parallel platforms mean doubled operational attention, and someone had to decide, on a schedule, when the old system dies.

04 Discovery and Diagnosis

Emerson Technology began with a focused discovery phase that treated the monolith as evidence rather than as an enemy. The team instrumented the legacy system to observe real traffic, then used version-control history to identify which modules changed most frequently, which modules failed most often in production, and where the two overlapped. That analysis cut through a year of abstract architecture debate: payments, leasing workflows and maintenance dispatch were simultaneously the highest-change, highest-failure, highest-revenue modules, which made them the obvious first targets for rebuild. Modules that had barely shipped in years — certain reporting screens, legacy admin tools — were explicitly deferred, because rebuilding them early would spend scarce attention on code that rarely mattered.

The second diagnostic thread was behavioral pinning. Before any rewrite, the team wrote contract and characterization tests around the legacy seams that the first new services would touch, capturing what the old system actually did — including its edge cases and its bugs — so the new implementations could be compared honestly against observed behavior rather than against documentation that had drifted from reality. The discovery phase also surfaced the operational constraints that any solution had to respect: rent-collection runs could not move until their replacement had survived several cycles of shadow traffic, and the single VPS meant there was no staging environment, so preview infrastructure had to be part of the new platform's foundation rather than an afterthought. The output was a sequenced migration plan with business value, not architectural elegance, as the ordering principle.

05 The Solution

The rebuild followed a module-by-module strangler migration rather than a big-bang rewrite: the monolith kept running every workflow it owned until its replacement had proven itself against live traffic, and the legacy PHP codebase was switched off nine months after kickoff. The new platform was deliberately conventional, because the goal was velocity and hire-ability, not novelty.

The new stack

The tenant and owner portal is a Next.js application backed by Node.js services running on AWS ECS, with PostgreSQL on Amazon RDS as the system of record. Redis-backed queues absorb the sharp, predictable bursts — payment runs on collection days and notification storms when maintenance statuses change — so a spike in one workflow no longer starves every other. Shared business logic moved into typed service modules with contract tests at their boundaries, which is what lets teams change a payments rule without re-reading the leasing codebase. The stack was chosen for boringness on purpose: Next.js, Node and Postgres are technologies the local hiring market knows deeply, which addressed the recruiting failure directly.

The path to production

Terraform and GitHub Actions gave every feature branch its own preview environment and an automated path to production, ending the practice of QA-ing on a shared server that everyone's changes contaminated. Structured logging and Sentry tracing made failures diagnosable in minutes rather than archaeology sessions, and DORA metrics — lead time, deployment frequency, change failure rate — were published internally so improvement was

measured rather than asserted. This delivery machinery mattered as much as the application code: a rebuild that lands on the same manual deployment habits simply reproduces the old pain with newer frameworks.

Shadow-traffic validation

Each migrated module was validated against shadow traffic from the monolith before cutover: real production requests were mirrored to the new service, and its responses were compared to the legacy behavior the contract tests had pinned down. Payments, leasing workflows and maintenance dispatch — the modules where velocity mattered most — went first under this discipline. Shadow comparison caught integration gaps that even careful test plans consistently miss: a tax-calculation rounding difference in billing, an edge case in how legacy code had handled prorated leases. Each was fixed before customers ever saw the new path, and each would have been a support emergency under a big-bang cutover.

06 The First 90 Days

The first month was spent almost entirely on foundations and pinning. Contract tests were written around the payments and leasing seams, the Terraform baseline for ECS, RDS and networking was reviewed and merged, and the preview-environment pipeline was stood up so that every subsequent line of code shipped through the new machinery. A named triage owner took responsibility for all legacy fixes during this period — a deliberately unglamorous role that shielded the rebuild team from the constant feature firefights that quietly double every migration timeline it touches. The team also announced, in week three, a sunset date for the legacy stack and began tracking monolith usage on a visible burn-down chart, converting an open-ended migration into a project with a deadline.

Weeks five through eight delivered the first production service: the payments module, running behind a feature flag while shadow traffic validated it against the monolith. In week nine the first cutover happened, and rent-collection day the following Monday was the real exam — it passed without an outage, the first clean collection day in memory. Weeks ten through twelve extended the pattern to leasing workflows, with weekly cutovers replacing the monthly ritual, and lead-time metrics published where the whole company could see them. By day 90, three of the five priority modules were live on the new platform, the deployment frequency graph had already inflected upward, and — the metric the leadership team watched most closely — the two enterprise prospects that had walked away accepted renewed demonstrations.

07 The Results

Feature velocity tripled. The team now ships more than 30 production changes a month against single digits before, and lead time from idea to production fell from three months to under two weeks. Portal page loads dropped 70% because the Next.js front end replaced a jQuery application never designed for performance, and mobile application completion rose 35% as the flow finally worked on the devices tenants actually use. Rent-collection days have run clean for a full year — the predictable outages are now stories new employees hear rather than nights they live. The two lost RFP accounts returned as customers within two quarters, and engineering attrition dropped to zero once the legacy stack, and the 2 a.m. cron alarms, were retired.

METRIC	BEFORE	AFTER
Feature velocity	Baseline	3x
Lead time to production	3 months	Under 2 weeks
Portal page loads	Legacy baseline	-70%
Production changes/month	8	30+

The less quantifiable changes compound the headline numbers. Product managers stopped batching features into large, risky releases because shipping small became cheaper than shipping big. Engineers began trusting the codebase — the word that appeared in the follow-up hiring interviews — because contract tests and preview environments made changes reversible and visible. And the roadmap conversation changed character entirely: once velocity tripled, the constraint on strategy stopped being what the platform could absorb and became what the market wanted next, which is the position a product company wants to be in.

08 What Your Business Should Do

If your platform is aging but your business is live, the Fairbrick sequence transfers almost directly. Migrate by module, not by big bang: rebuild the highest-value workflows first and route shadow traffic through them before cutover, because parallel-run validation catches integration gaps that even careful test plans consistently miss. Add tests to the legacy seams before you rewrite anything — contract and characterization tests pin down what the old system actually does and prevent the expensive surprises that sink rewrites. Choose boring, mainstream technology that your local hiring market knows deeply; exotic choices compound recruiting pain for years after launch day. Then build the golden path to production — automated build, test and deploy so every merge can ship safely — because a one-command path is what removes the friction that pushes teams toward batching risky changes.

Run the migration as a business program, not a science project. Sequence the work by revenue impact and change frequency rather than by architectural elegance, and let the velocity gains from early modules fund patience and goodwill for the rest. Set a deadline for the legacy stack, announce it internally, and track usage on a visible burn-down chart — open-ended migrations drift for years, while real dates force decommission decisions. Measure velocity publicly: publish lead time, deployment frequency and change failure rate where the whole company can see them, which both sustains momentum and attracts exactly the engineers you need. Finally, protect the first 90 days of focus by naming a clear triage owner for legacy fixes; constant context switching quietly doubles every migration timeline it touches. While you plan, hold your own team to the questions this engagement raised: what fraction of your sprints maintains systems nobody would choose today, which two modules would unlock the most customer-visible velocity if rebuilt first, how long you could tolerate two platforms in parallel and who decides, and which deals you have lost to a faster competitor this year. Work through these eight steps with your team using the action checklist in this practice book, and revisit the reflection questions each quarter so the rebuild keeps serving the business instead of becoming a monument.

REAL-WORLD PROOF & BENCHMARKS

McKinsey research reports that companies in the top quartile of developer velocity achieve revenue growth up to five times faster than bottom-quartile peers. - McKinsey, 2020

DORA research consistently links stronger software delivery performance to better overall organizational outcomes, including profitability and market share, across thousands of teams studied. - DORA Report

Stripe's Developer Coefficient survey reportedly found developers spending a third or more of their time on technical debt and maintenance rather than new features. - Stripe, 2018

THE PRACTICE PAGES

The 8-action checklist

Work through the list with your team. Tick each box as your organization completes the step, and fill in an owner and a target date so every action has a name and a deadline.

☐

1. Migrate by module, not big bang

Rebuild the highest-value workflows first and route shadow traffic through them before cutover. Parallel-run validation catches integration gaps that even careful test plans consistently miss.

OWNER

TARGET DATE

STATUS

--	--	--

☐

2. Add tests to legacy seams first

Pin down current behavior with contract and characterization tests before rewriting anything. Knowing precisely what the old system does prevents the expensive surprises that sink rewrites.

OWNER

TARGET DATE

STATUS

--	--	--

☐

3. Choose boring, mainstream technology

Pick stacks your local hiring market knows deeply, such as Next.js, Node and Postgres. Exotic technology choices compound recruiting pain for years after launch day.

OWNER

TARGET DATE

STATUS

--	--	--

☐

4. Build the golden path to production

Automate build, test and deploy so every merge can ship safely. A one-command path to production removes the friction that pushes teams toward batching risky changes.

OWNER

TARGET DATE

STATUS

--	--	--

☐

5. Sequence migration by business value

Rank modules by revenue impact and change frequency, then migrate in that order. Velocity gains from the early modules fund patience and goodwill for the rest.

OWNER

TARGET DATE

STATUS

--	--	--

☐

6. Set a deadline for the legacy stack

Announce a sunset date for the legacy system and track its usage on a visible burn-down chart. Open-ended migrations drift for years; real dates force decommission decisions.

OWNER

TARGET DATE

STATUS

--	--	--

☐

7. Measure velocity publicly

Publish lead time, deployment frequency and change failure rate where the whole company can see them. Transparent delivery metrics also attract exactly the engineers you need.

OWNER

TARGET DATE

STATUS

--	--	--

☐

8. Protect the first 90 days of focus

Shield the rebuild team from feature firefights by naming a clear triage owner for legacy fixes. Constant context switching quietly doubles every migration timeline it touches.

OWNER

TARGET DATE

STATUS

--	--	--

AFTER THE CHECKLIST: OUR TOP 3 PRIORITIES

From the eight actions above, write down the three you will start first - and the single number you expect each one to move. Three priorities with owners beat eight without any.

PRIORITY 1 NUMBER IT SHOULD MOVE	PRIORITY 2 NUMBER IT SHOULD MOVE	PRIORITY 3 NUMBER IT SHOULD MOVE
--	--	--

THE PRACTICE PAGES

Reflections for your team

Use these questions in your next planning session and type short answers into the boxes. The gaps you find are usually the fastest wins.

Q1. What fraction of your engineering sprints goes to maintaining systems nobody would choose today?

Q2. Which two modules, if rebuilt first, would unlock the most customer-visible velocity?

Q3. How long could your business tolerate two platforms running in parallel — and who decides?

Q4. What would your engineers say if asked whether they trust the current codebase?

Q5. Which customer deals have you lost to a competitor's faster product in the last year?

Q6. If velocity tripled next quarter, what would your roadmap ask for first?

THE PRACTICE PAGES

My 90-day plan

Finish the session by committing to a plan. Three phases, one goal each, one number that tells you it worked. Bring this page to your next leadership review - or to a free consultation with Emerson Technology, and we will pressure-test it with you.

Days 1-30 Focus: audit and alignment

PRIMARY GOAL

KEY ACTIONS

HOW WE MEASURE SUCCESS

OWNER

Days 31-60 Focus: build and prove

PRIMARY GOAL

KEY ACTIONS

HOW WE MEASURE SUCCESS

OWNER

Days 61-90 Focus: scale and measure

PRIMARY GOAL

KEY ACTIONS

HOW WE MEASURE SUCCESS

OWNER

Tip: save a copy per team so each group keeps its own answers. Revisit the reflections every quarter - the questions stay the same even as the numbers move.

BOOK THE REVIEW BEFORE YOU CLOSE THE BOOK

Plans without a review date drift. Before you close this practice book, agree on when you will come back to these pages, who will run that session, and when you will check the numbers again. If you want an outside pair of eyes, bring this page to a free consultation with Emerson Technology and we will pressure-test the plan with you.

REVIEW DATE	REVIEWED BY	NEXT CHECK-IN
------------------------------------	------------------------------------	--------------------------------------

Ready for a transformation like Fairbrick Living's?

Emerson Technology designs and builds the data, AI, cloud and automation systems behind results like these. On a free consultation call we will map this exact playbook to your business - your data, your constraints, your numbers.

Book a free consultation

support@emersontechnology.com

or start online: emersontechnology.com/contact