



# Legacy, but critical: A playbook to evolve core systems.

Prachi Tyagi, Lead Consultant.

 **thoughtworks**

Design. Engineering. AI.

<b>A closer look at the system</b>	<b>4</b>
<b>The landscape</b>	<b>5</b>
<b>From story to runbook: Four themes for working in legacy</b>	<b>12</b>
<b>Closing thoughts</b>	<b>16</b>
<b>About the author</b>	<b>17</b>

## **The legacy evolution playbook.**

Modernizing business-critical systems isn't about "turning them off and rewriting." It's a delicate balancing act of keeping decades-old revenue-generating engines running while preparing for the future.

Thoughtworks helps enterprises rebuild the systems their business depends on, without stopping the business to do it. Based on real-world experience, this playbook outlines a strategy to make systems visible, changes safe and teams more collaborative, focusing on high-impact "targeted surgery" rather than invasive overhauls.

### **Lessons from a sports tech client.**

In sports tech, everyone loves talking about next-gen platforms: low-latency data pipelines, real-time analytics and cloud-native architectures. But for most organizations, the systems that make the money today aren't greenfield. They're long-lived platforms that have been adapted, extended and patched over many years to support new sports, new markets, new regulations and new customers. In sports betting, those legacy platforms are often:

- Always-on and real-time: events are happening every second, markets move and prices change.
- Directly tied to revenue: if the system can't calculate and update the prices for different outcomes, bets can't be placed.
- Deeply integrated: dozens of upstream feeds and downstream consumers depend on them.

You can't just "turn them off and rewrite." You have to keep them running, keep them safe and keep them evolving, even while the "next-gen" platform is being designed.

**This playbook is about one of those systems.**

## **A closer look at the system.**

For the last couple of years, I've been working with a client in the sports-betting space on a large, business-critical legacy platform. Roughly speaking, this system:

- Ingests sports data for live matches.
- Produces prices and markets that traders and automated processes can work with.
- Exposes that information so bettors can place in-play bets through partner bookmakers.

In short, it's a long-lived legacy engine (born on-prem over a decade ago) that has since been migrated onto AWS, using services such as EKS, MSK, Lambda and more to decouple it from its original architecture. If this engine stops doing its job, traders can't offer the right markets, bookmakers can't take bets, and a big part of the live betting experience simply doesn't exist. So even though it's "legacy," it's not a nice-to-have. It's a profit-making, always-on engine in the middle of the sports-betting business.

When our team from Thoughtworks joined, the legacy engine had already been running in production for years. Around it, the client had also started a modernization stream to design a new platform that would eventually take over this legacy. So our work sat right in the middle of that picture. Keeping this legacy platform healthy today, making it clearer and safer to change and making sensible decisions about what still made sense to do in legacy while a rewrite happens in a modernization stream.

This playbook is about what we found, what we changed and the practices we'd reuse on any tech legacy system. Consider this as a menu, not a checklist, and select the parts that best fit your context.

## The landscape.

### 1. Knowledge and documentation.

The system worked. It was making money, but changing anything required a long lead time and interlinking with other components meant implementing any change was trial and error. Understanding it was a different story:

- **Knowledge silos:** critical flows lived in people's heads and scattered chats.
- **No single product view:** no clear "what is this platform, who uses it, where does it sit in the ecosystem?"
- **Sparse / fragmented documentation:** bits of architecture and process docs existed, but were outdated or hard to find.
- **Ad-hoc knowledge transfer:** knowledge transfer happened in one-off meetings with no reusable artifacts.
- **Domain rules buried in code:** business logic was only visible in 10+ year old classes.
- **Complex release process in people's minds:** the release "runbook" was tribal knowledge and hard to follow for new joiners.

## 2. Ways of working.

On the surface, the team was “agile.” In reality:

- Ways of working weren’t explicit: ceremonies existed, but their purpose wasn’t clear.
- There was no shared “definition of done” for this system.
- There was no clear way to prioritize the backlog.
- The path to production wasn’t written down; it lived in a few people’s heads.

## 3. Local setup and safety.

From a technical safety perspective:

- **Slow, painful setup:** the system’s local setup was tightly coupled to a very specific type of machine, so getting it running on a new laptop was one of our biggest challenges. New engineers often took months to get a stable local environment.
- **Limited tests where it mattered:** critical flows weren’t adequately protected.
- **Tight coupling in code and ownership:** certain components were both technically complex and socially “owned” by specific individuals.

### **Pro tip:** The “new joiner” metric

Measure the health of your legacy system by how many days it takes a new engineer to achieve their first “useful” local environment. If it takes weeks, your biggest risk isn’t the code, it’s the onboarding bottleneck.



#### 4. Team dynamics.

- **Team composition:** the team was a blend of long-tenured client engineers, experts who had carried the critical production system for 10 – 20 years and a newly integrated Thoughtworks team.
- **Knowledge transfer:** the client engineers held the deepest, most valuable system knowledge. Our focus was on developing a collaborative approach to capturing and documenting this context, being mindful of their current workload and the historical complexity of the platform.

#### 5. Scaling a system that will eventually be replaced.

Finally, the product side wanted to onboard more bookmakers and traffic. The legacy platform had performance bottlenecks under peak load. The modernization stream would eventually take over, but not within the year. We needed to improve performance and resilience without starting a six to seven month mini-modernization inside legacy.



## What we changed (in practice).

### Making knowledge visible.

To make the system understandable without always “calling a friend,” we:

- Created a “helicopter view” of the platform: what it does, who uses it, key user journeys and upstream/downstream systems as both docs and a short recording.
- Wrote architecture overviews for key components and features.
- Introduced architecture decision records (ADRs) to capture “why we chose X over Y” on important decisions.
- Built a domain glossary so code, tickets and conversations used the same language.
- Captured main use cases in simple flows and demo recordings.
- Started a lessons-learned repository of everyday discoveries and gotchas.
- Held a weekly “what we learned” session and recorded knowledge transfer sessions, indexing them in one place.

We also leaned heavily on AI-assisted tools. We also leaned heavily on AI-assisted tools (Copilot, Cursor, and Junie):

- Asking them to explain what happens when a particular event or request hits the system.
- Generating sequence diagrams and class-level diagrams for complex flows.
- Reviewing and then storing those outputs in our docs so they became team assets, not just transient chat history.

## **Making ways of working explicit.**

We didn't drop a new process on the team. Instead, we observed and then made the implicit explicit: Co-created a ways of working document with the existing team that captured:

- Ceremonies and their purpose.
- Core working hours and communication channels.
- How support and incidents are handled.
- How technical decisions are made.
- Defined and documented the path to production for this system.
- "Definition of done" was clearly determined.
- A step-by-step release guide including verification and rollback was created.
- Documented deployment strategies across old and new components.

## **Improving setup and safety nets.**

To make change safer:

- We dramatically improved the setup with scripts, prerequisites, seed data and troubleshooting notes, making it work on a wider range of machines and reducing the time to a "first useful environment" from months to just a couple of days.
- We focused on "testing where it matters": adding unit and integration tests around critical flows, plus performance tests where appropriate.
- We used pairing intentionally, especially on support tickets and dauntingly complex modules, to spread context and increase the bus factor.

## Changing how we showed up in conversations.

The biggest shift was not in code, but in attitude:

- We moved from blame (“they don’t document”) to empathy (“what constraints are they under?”). Building relationships with stakeholders outside the team helped accelerate this shift, because we saw the gaps weren’t “this team’s problem,” but a broader, company-wide culture and set of constraints.
- We matched people’s preferred communication styles: 1:1 conversations, small calls or chat-based walkthroughs.
- We made it clear we were on the same side, attached to the shared mission (keeping the system healthy), not to “our favorite way of working.”
- Where possible, we offered: “If you talk us through it once, we’ll write it up and you just correct it.”

### **Pro tip:** Use “targeted empathy”

When dealing with experts who have managed a system for 20 years, don’t ask for documentation. Ask to “shadow” them or offer: “If you talk us through it once, we’ll write it up and you just correct it.” It lowers their cognitive load while capturing tribal knowledge.



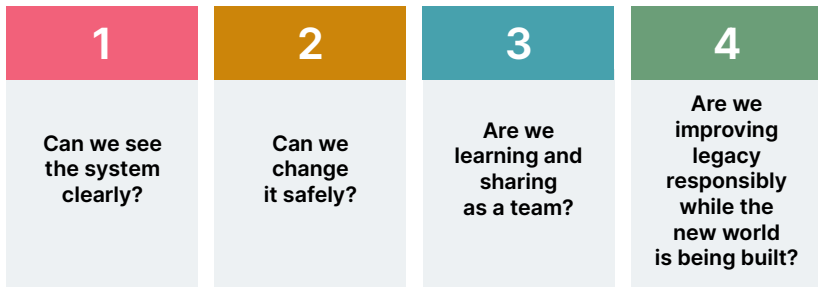
## **Scaling without rewriting.**

On the scaling side, we treated improvements as targeted surgery, not a heart transplant:

- First, we added metrics and observability around the parts of the pipeline that were under pressure.
- We studied real behavior under load, not just averages.
- We brainstormed options, but favored small, targeted changes with high impact and low overlap with the future architecture. For example, we right-sized capacity (database, message queues, CPUs/pods), removed hotspots by shifting load away from an overloaded database and tweaked the partition strategy of the message queue. We continue to make similar focused improvements, such as routing non-critical messages to a separate queue, running expensive sports in a dedicated deployment. These remain incremental, targeted steps rather than a large re-architecture.
- We constantly asked: “What is the smallest meaningful change that moves this bottleneck, without starting a hidden rewrite?” This enabled us to increase resilience and throughput for today’s system, while keeping the real re-architecture work in the modernization stream, where it belongs.

## From story to runbook: Four themes for working in legacy.

Out of this journey, a lightweight legacy playbook started to emerge. It's not a step-by-step recipe, but more of a set of themes that can be reused when joining any legacy system. The legacy playbook can be structured around four core questions:



### Theme 1 – Make the system visible: product, code and flow.

A lot of the early pain comes from the fact that the system works, but is hard to see:

- No single product view.
- Knowledge silos.
- Domain rules buried deep in code.

Our first theme can be about making the system visible:

- Create a helicopter view of the product: what it does, who uses it, key flows, upstream/downstream systems, and keep it simple enough that new joiners can orient in an hour, not a month.
- Build a shared language: a domain glossary and a few documented use cases so code, tickets and conversations align.

- Use AI-assisted tools to map the code: ask them to explain flows, generate sequence/class diagrams, summarize modules, then curate and store those artefacts as part of the documentation, not just in chat.
- Capture every day discoveries in lessons-learned notes and indexed knowledge transfer recordings so the next person does not start from zero

*The goal isn't perfect documentation; it's a shared, lightweight picture of "what this thing really is" that everyone can work from.*

## **Theme 2 – Make change safe: how we run, release and protect critical flows.**

The second theme is about being able to touch the system without flinching. That means:

- Having at least one reliable way to run and debug the system, locally or in a shared environment, with clear prerequisites, scripts/steps and a simple smoke test.
- Making the path to production explicit: which environments exist, who deploys where, what needs to be true before we release and how we verify and roll back.
- Agreeing a realistic "definition of done" for this system (not a generic one): reviews, tests where they matter, monitoring for risky changes, docs updated when it's useful.
- Protecting critical flows (the ones that really hurt when they fail) with focused tests, better logging and basic metrics.

*We can follow a simple rule: if we touch a critical flow, we leave its tests and observability slightly better than we found them - making the system safer to change, release and support with confidence.*

### **Theme 3 – Build a learning, shared-ownership team.**

Legacy work is as much about people as it is about code. We can see that the combination of mixed tenure, opaque history and unclear ways of working could easily create friction. This theme is about turning that into shared ownership rather than “us vs them”. Key elements:

- Make ways of working explicitly together: ceremonies and their purpose, how work enters the team, how support and incidents are handled, core hours and communication channels, how technical decisions are made and recorded (e.g. architecture decision records).
- Create lightweight loops of learning: a regular “what we learned this week” slot and support playbooks for recurring incidents (symptoms, repro, root cause, fix, follow-up).
- Approach long-tenured engineers with empathy, not blame.
- Match their preferred communication style (1:1, small calls, chat).
- Offer “you talk, we document and bring it back for review”.

*This single guiding principle can shape our internal approach: don't get attached to your favorite way of working, get attached to the shared mission. This theme can turn “their system” into “our system”.*

### **Theme 4 – Improve legacy responsibly when a rewrite is around the corner.**

The last theme is the balancing act we have lived every day: how far do we go in legacy when a new platform is being built? The legacy platform still needs fixes, improvements and resilience. But we don't want to quietly build “version 2.0” inside the old codebase.



## **Pro tip: A decision lens for legacy changes.**

Use these four questions before starting any non-trivial change:

### **1. Time horizon.**

- Will this change deliver real value in the next three to six months, while the legacy system has a longer lifespan?

### **2. Scope – targeted surgery, not a heart transplant.**

- Can we do this as small, incremental changes in a limited area?
- Or does it require a big redesign across multiple modules, teams or data models?

### **3. Overlap with the new system.**

- Is the new platform likely to build the same capability?
- If yes, can our legacy work be reused in the new platform (shared component/service, Application Programming Interface contract)
- If it cannot be reused, are we at risk of building the same thing twice or can it be reused in some way in the new platform?

### **4. Risk vs reward.**

- Does this change remove real pain (incidents, confusion, operational load)? Or is it mainly about making the design “prettier” without clear short-term benefit?

These questions can help us decide what belongs in legacy and what should be treated as an input to the modernization backlog instead.

*The principle is simple: We improve legacy when the change is near-term, targeted, and either low-overlap or reusable; if it is broad, invasive, and strongly overlapped with the future design with no clear reuse, it belongs in modernization.*

## **Closing thoughts.**

Legacy systems are often where the real money is made. They carry years of decisions, edge cases and scars from live operations. Working in that world is less about “clean code” and more about:

- Making knowledge and flows visible.
- Making today's change safer than yesterday's.
- Improving just enough to meet real demand.
- And being kind to the people who've kept the system alive for years.

If we treat legacy work as stabilizing, enhancing and improving without doing a rewrite, we create resilience in the system and in the team, while giving the new platform the space to grow in the right direction.

## About the author.



### **Prachi Tyagi**

**Lead Consultant, Thoughtworks.**

With 16+ years in software engineering, Prachi has worked on distributed systems and the modernization of complex legacy platforms. Her background spans architecture, continuous delivery, testing, and scalable software design. More recently, she's been drawn to AI/ML, particularly MLOps and Generative AI, and how they can shape developer workflows, system design, and intelligent software systems.

[Connect with Prachi on LinkedIn](#)

### **Contributions.**

Finally, a heartfelt thanks to Jas Baweja, Thomas Wolle and Silke Hofmann for their meaningful contributions and steady partnership throughout the project. Their collaboration and support were key in helping us address the challenges of evolving a business-critical legacy platform safely.

We are a global technology consultancy that delivers extraordinary impact by blending design, engineering and AI expertise.

For over 30 years, our culture of innovation and technological excellence has helped clients strengthen their enterprise systems, scale with agility and create seamless digital experiences.

We're dedicated to solving our clients' most critical challenges, combining AI and human ingenuity to turn their ambitious ideas into reality.

[thoughtworks.com](https://www.thoughtworks.com)