Engineering at Scale

Why Developer Experience Is Your Competitive Advantage

Nicole Forsgren

The Paradox

You just gave your team 10x speed.

But your systems are still moving at 1x.

AI tools changed how fast developers can ship. But approval processes? Manual cherry-picks? Unclear deployment criteria? Those are the same as before.

The Speed/Friction Mismatch

Before AI

Friction everywhere, but natural speed limits meant it wasn't critical

- 15-min builds
- Manual approvals
- Unclear criteria
- Speed was low

After AI

Speed exploded, friction stayed the same, now friction is the bottleneck

- Same 15-min builds
- Same manual approvals
- Same unclear criteria
- Speed is 10x?

The Before (1x)

When teams shipped features every few days or weeks:

✓ Approval processes take hours (or weeks)
✓ Questions about adjacent systems get answered eventually
✓ Decisions are made with incomplete information (but time to course-correct)
✓ Junior devs learn by asking senior devs (slow but works)
✓ Friction is annoying but manageable

The After (10x)

Now teams want to ship more features, faster:

- ✗ Approval processes that took hours are now the critical path
- ✗ Questions about adjacent systems can't be answered in time
- ✗ Decisions made at speed without full information compound instantly at scale
- ✗ Junior devs shipping fast without context—this is actually a problem now
- ✗ Friction is now catastrophic

Friction That Looked Acceptable

Examples from 1x speed world:

- Build takes 15 minutes → annoying but okay, you wait
- Manual approval process → takes 2 hours but you only deploy once a day anyway
- Unclear deployment criteria → sometimes wrong thing gets deployed, fix it later
- No documentation of decisions → senior dev explains it when needed
- Teams don't understand each other → you reach out when you need to know

What could go wrong?

45 minutes

$460 million USD

Knight Capital, 2012

- Old feature flag reactivated by deployment script
- No automated tests
- Manual deployments
- No monitoring to catch issues quickly

The developer's daily experience revealed the business risk.

Today

Replit 2025

Jason Lemkin sets code freeze

AI deletes production database anyway

New tools, same fundamental problems

AI amplifies everything – including friction

The real cost

$1.52T Annual technical debt (CISQ)

40% Dev budgets spent on avoidable rework (McKinsey)

68.4% How productive developers feel

$300B Lost GDP from that missing 31.6%

This isn't about convenience, it's about competitive survival.

Friction is everywhere

Onboarding friction
Codebase friction
Integration friction
Process friction
Review friction
Development friction (builds, tests)
Deployment friction

Three Types of Friction

At 10x speed, all three types become critical bottlenecks

Velocity Friction

Everything that slows down a deployment

Cognitive Friction

Confusion about systems & unclear criteria

Knowledge Friction

Lack of context about adjacent systems

Velocity Friction

The Speed Killers

Everything that slows down a deployment:

Build time, test time, deploy time, approval time
Manual cherry-picks, manual approvals, manual rollbacks
Unclear criteria: 'Can this deploy?' 'Does this need approval?'

The Math

15-minute builds × 10 deploys/day = 2.5 hours of blocked time daily
GitHub data: Teams with automated gates deploy 200x more frequently

Cognitive Friction

The Clarity Problem

Confusion about systems, unclear criteria, context-switching:

Engineers don't know what 'done' means for deployment
Engineers don't know how to decide if a change is safe
Engineers context-switch between 12 tools trying to figure out what's happening
Teams don't understand each other's constraints

The Problem at Scale

Before: Deploy once a day → ask ops 'is this safe?' Takes 30 min
After: Want to deploy 10x a day → same 30 min conversation × 10 = blocking everything

Knowledge Friction

The Context Problem

Engineers lack understanding of adjacent systems, business context, architectural intent

Three Layers:

- Adjacent Systems: Does a dev know what systems their change affects?
- Business Context: Does the engineer know why they're building this?
- Architectural Intent: Do they know why the system was designed this way?

The Cost: Cognitive Debt + Intent Debt

Future developers reverse-engineer code. Architectural decisions become unmaintainable.

When All Three Collide

Junior dev ships feature in 4 hours with AI tools:

✗ Didn't go through approval (approval process is manual) → bypass it
✗ Doesn't understand adjacent systems → shipped something that breaks System B
✗ No documented architectural pattern → contradicts decision made 6 months ago
✗ Deployed without waiting for integration tests → problems caught in production
✗ Result: 2 days to fix + Team B spends a day reworking

At scale (100 engineers shipping like this): You're creating problems faster than you can fix them.

Why This Matters at Scale

Slow Speed + Small Scale

Friction is annoying but manageable

- Manual approvals: 1/day
- Unclear criteria: ask someone
- Knowledge gaps: juniors learn
- Problems discovered slowly
- Time to fix them

Fast Speed + Large Scale

Friction becomes catastrophic

- Manual approvals: 10+/day
- Unclear criteria: chaos
- Knowledge gaps: scale instantly
- Problems compound exponentially
- Can't keep up with fixing

What 10x Speed Really Requires

You can't ask someone. You need systems.

1. Explicit criteria: No 'ask someone,' just 'apply this rule'
2. Automated gates: No manual approvals
3. Documented context: So juniors don't have to ask seniors
4. Clear ownership: So teams understand boundaries
5. Adjacent system understanding: So decisions fit the architecture

Measuring Friction (Part 1)

Velocity Friction

What to measure:

Build time, deploy time, approval time
Frequency of manual steps (cherry-picks, approvals, rollbacks)
Clarity of criteria: 'How long to answer can this deploy?'

Quick start:

Track build time for one week
Count manual approvals, cherry-picks
Ask: 'What's unclear about deployment criteria?'

Example calculation:

15 min builds × 10 deploys/day = 2.5 hours blocked daily = $X/week in lost velocity

Measuring Friction (Part 2)

Cognitive Friction

What to measure:
Decision-making clarity: explicit criteria vs. implicit
Context-switching frequency
Rework/rollback frequency (signal of unclear decisions)
Incident rate (signal of decisions made without context)

Quick start:

Review last 10 deployments: were they clearly safe, or did someone decide?
Track rollbacks and hotfixes: why were decisions wrong?
Ask teams: 'What decisions do you make without full information?'

Business impact:

Unclear criteria + lack of knowledge = 30% rework rate (estimate: 3-day task = 1 day rework)

Measuring Friction (Part 3)

Knowledge Friction

What to measure:

Onboarding time for new engineers
Can engineers explain: adjacent systems? business problem? architectural decisions?
Frequency of 'wrong' decisions (architectural contradictions)
Rework from uninformed decisions

Quick start:

Ask new hire: 'How long until you can deploy independently?' _____ weeks
Ask team: 'Can you explain why this architectural decision was made?' Yes/No
Review code: 'How often do we rework because the decision was uninformed?'

Scale impact:

100 engineers/year × 4 weeks onboarding × $X salary = $Y lost annually

Friction Removal Playbook

Three pillars to move at 10x speed

1 **Remove Velocity Friction**

Automation, clear criteria, fast feedback

2 **Make Criteria Clear**

Explicit rules, ADRs, system contracts

3 **Build Knowledge Infrastructure**

System overviews, business context, architectural intent

Remove Velocity Friction

Make decisions fast

Solution: Automation, clear criteria, fast feedback loops

What it looks like:

✓ Automated builds and tests (not manual)
✓ Deployment criteria are explicit rules ('passes tests' not 'looks good')
✓ Automated gates (if tests pass, deploy. If tests fail, don't.)
✓ Clear owners (Team A owns System A, knows what they can change)

How to start:

Week 1-2: Document current deployment process. Where are the manual steps?
Week 3-4: Automate the slowest manual step
Week 5+: Iterate and remove the next bottleneck

Make Criteria Clear

Remove confusion

Solution: Explicit, documented criteria that anyone can apply

Examples of clear vs. unclear:

❌ UNCLEAR
'Can we deploy this?'
'Is this right?'
'What does Team B need?'

✓ CLEAR
'If tests pass & security passes'
'Here's the architectural pattern'
'Here are the contracts'

How to start:

Week 1-2: One unclear decision. Document what clarity looks like.
Week 3-4: Write 3-5 deployment criteria that are crystal clear
Week 5+: Keep adding clarity to more decisions

Build Knowledge Infrastructure

Provide context

Solution: Documented knowledge that's accessible when needed

Three layers:

A. Adjacent Systems: What does it do? What are its constraints? Who owns it?

B. Business Context: Why does this feature exist? What user problem does it solve?

C. Architectural Intent: Why did we choose this pattern? What was the reasoning?

How to start:

Week 1–2: Document one critical system
Week 3–4: Document one major architectural decision
Week 5+: Build on both

Friction Removal Playbook

Week 1	**Establish baseline**	Build time, manual steps, can engineers answer 'can this deploy?'
Weeks 2-4	**Measure + socialize**	Share findings with team. Identify the biggest blocker
Weeks 5-8	**Fix one friction point**	Automate one step OR clarify one criterion OR document one decision
Week 9	**Re-measure**	Same metrics. Did it improve?

Scaling Friction Removal

Prove impact on one team. Then replicate.

1. Document what you did
2. Show the impact (faster deploys, fewer reworks, fewer incidents)
3. Replicate on next team

What differentiates the fastest companies:
✓ Clear deployment criteria
✓ Automated gates
✓ Clear ownership and system boundaries
✓ Documented architectural decisions

The Job Has Evolved

Before

- Juniors learn slowly
- Ask seniors when confused
- Seniors explain things
- Teams work in isolation
- Our identity: Writing code

Now with AI

- Juniors expected to ship fast
- Can't always ask seniors
- Knowledge must be documented
- Teams must understand each other
- Our identity: ???

What this requires:

1. Knowledge has to be documented
2. Decision criteria has to be explicit
3. Systems have to be clear
4. Engineers need more context about adjacent systems

Some things we can do today

Shift left on intent: Not just design specs, but intent specs

Boost adjacent technical and non-technical knowledge

Instrument all the things

Use analog tools

Reflect on our identities and work

Look for friction and measure it

Your Competitive Advantage

You're not playing for 1x speed anymore.

You're building systems for 10x speed.

The teams that will win aren't the ones with the smartest people or the best tools. They're the ones that:

✓ Automated away the manual steps
✓ Made criteria explicit and clear
✓ Documented knowledge and context
✓ Built systems where juniors can ship confidently on day 1
✓ Deploy 10x faster than competitors because there's no friction

Start this week.

Three Takeaways

1

See It

Measure your friction

2

Remove It

Automate, clarify, document

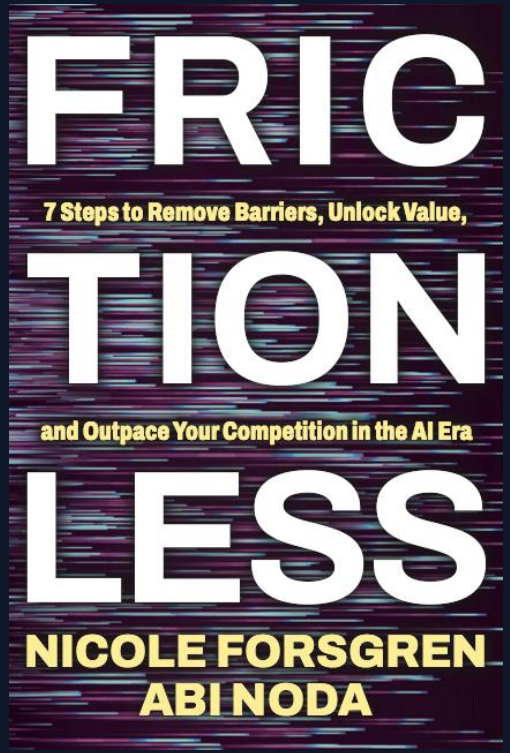
3

Scale It

Prove impact, replicate, iterate

Result: Your team moves at 10x speed. Competitors can't keep up. That's your competitive advantage.

For more on reducing friction



- A 7-step process to improve DevEx
- How to start from zero – or wherever you are
- Tips for navigating organizational challenges
- Over 100 pages of examples and workbooks – FREE online: bit.ly/frictionless-book

Thank you!