Now booting: LUISE.EXE

Boot Profile: Codemotion Milan 2025 Edition

Architecture: Azure

User Profile: Architect | MVP | Open Source Advocate | she/her

Visual Identity: #ff69b4

> Checking system components...

Creativity Engine.....[OK]

DevOps Pipeline Alignment.....[OK]

Accessibility Compliance Scanner.....[ENABLED]

Punk Attitude.....[ACTIVE]

Princess Protocols..... .....[👑 READY]

> Loading core values...

Motto: "Changing the world one app at a time ✨"

Favorite number: 42

Response model: "It depends!"

Documentation is her love language

> Initializing personal modules...

Hobbies: Running | Healthy Foodie | LEGO

Known aliases: Mother of Dragons (a.k.a. Mom of Teens)

Authenticating with recognition services...

Microsoft MVP since 2018

Website: m365princess.com

Federated ID: @LuiseFreese.bsky.social

> Finalizing launch...System ready.

LUISE.EXE is now running.

> Launching diagnostics...

Module detected: Your testing strategy is broken – let's fix it

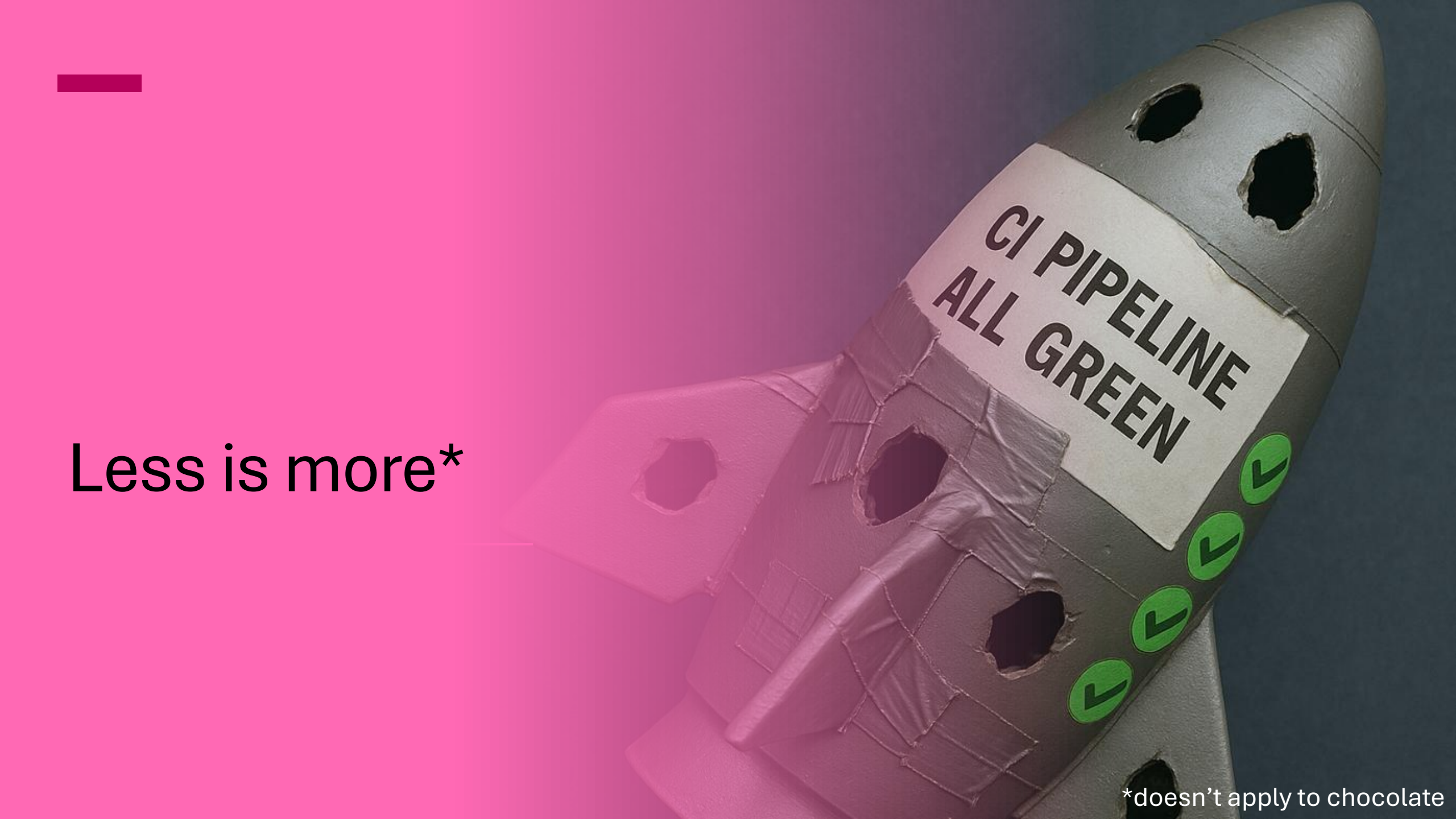
Status: Critical insight available

Recommended action: Attend session

We’ve all shipped bugs we were sure were covered
This session is a call to rethink why we test, what we test, and how we test

Test less!

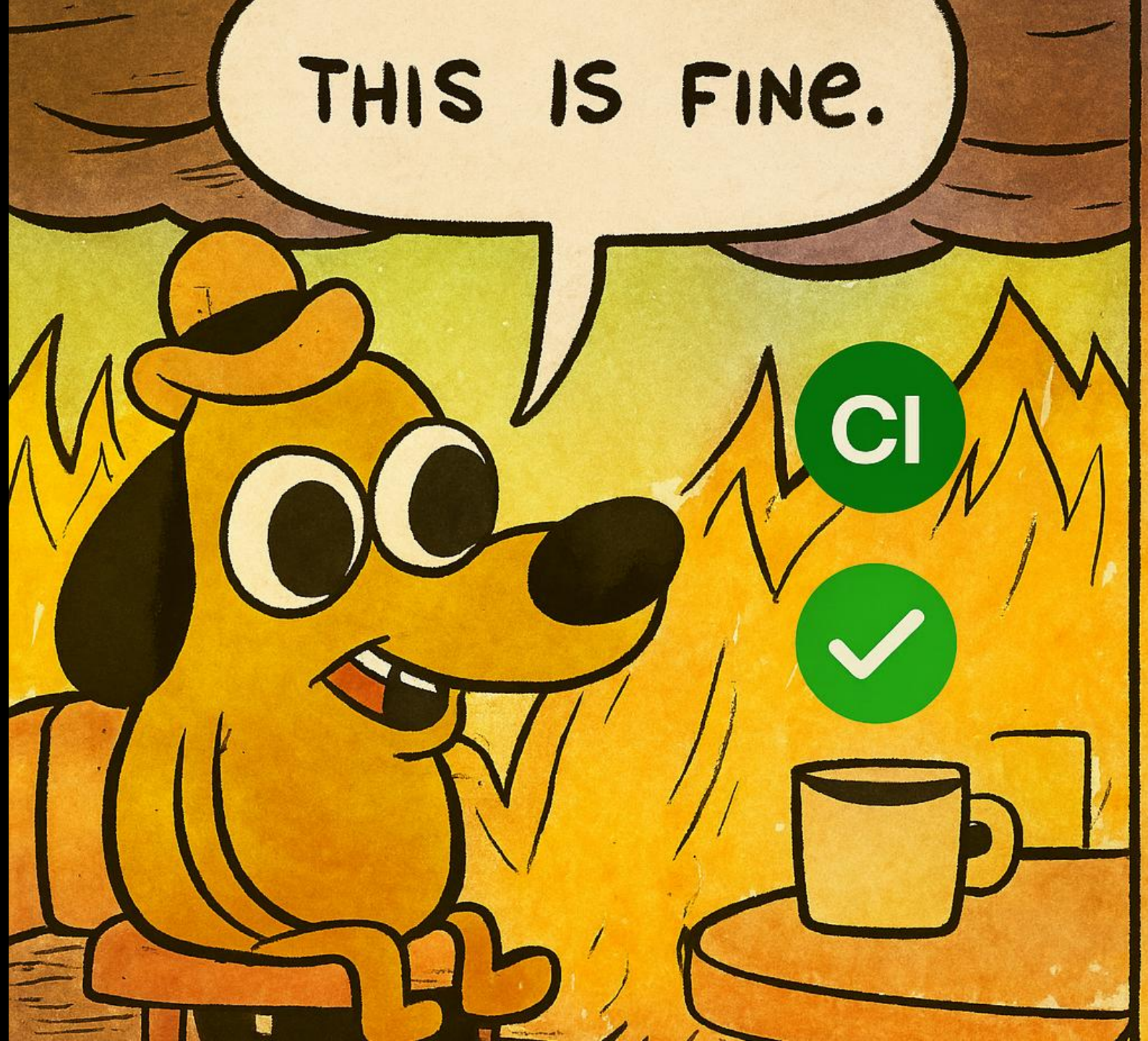
Less is more*



*doesn’t apply to chocolate

Sounds familiar?

- High test coverage
- All builds green
- Production’s on fire



Green ain’t safe.

- Coverage ain’t confidence.
- And deep in your heart, you know that.
- That’s the reason why you refuse to deploy on Fridays – ‘cause you can’t trust your tests.

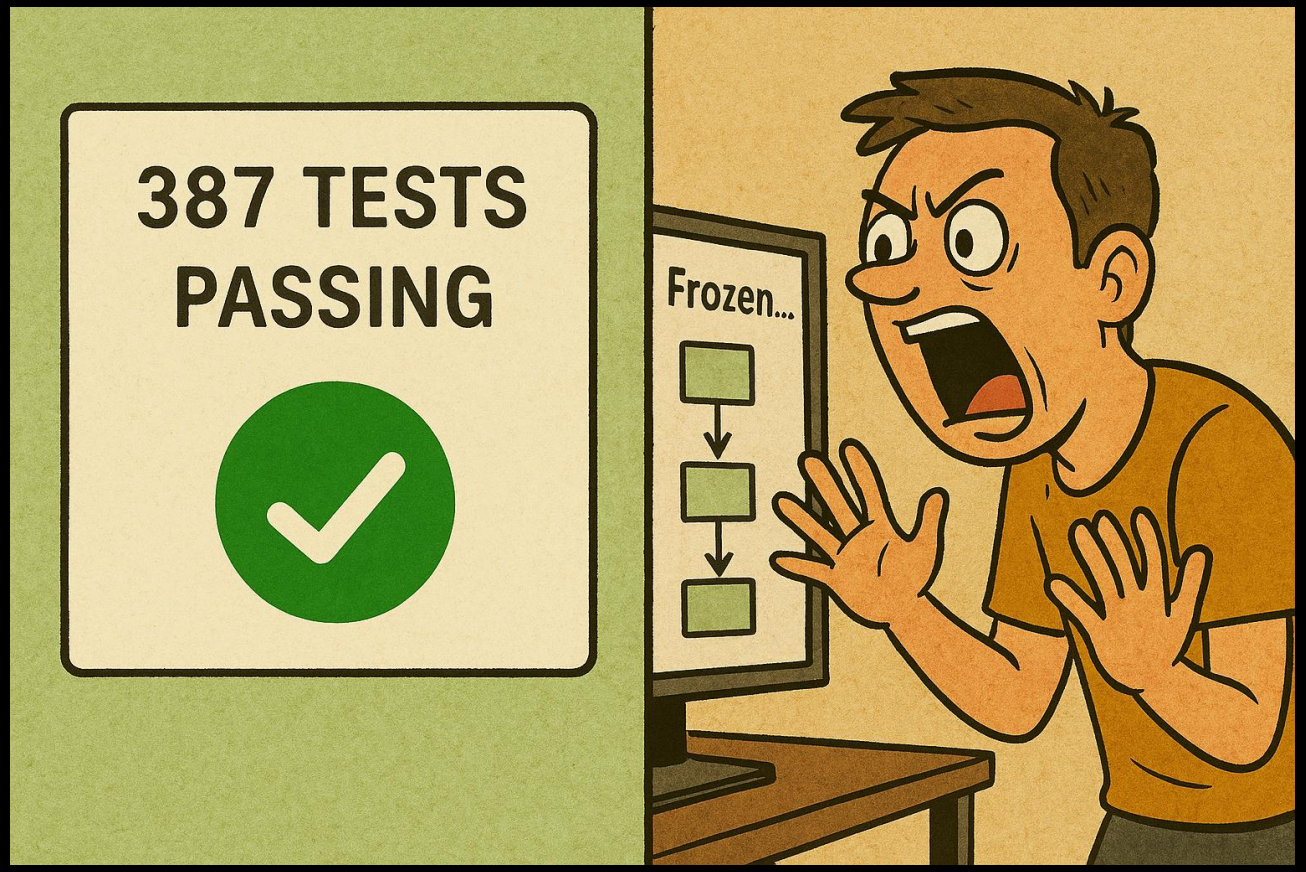
You created a system in which developers are afraid to ship

Your users don’t care how many tests you’ve written.

They care that your software works.

“Please don’t confuse test quantity with software reliability under pressure”

We test what we can count – not what really counts



- Testing is a means, not the end
- Too many teams treat test coverage like a performance metric
- In reality, it’s a vanity metric
- Doesn’t add value per se
- Doesn’t necessarily contribute to our ultimate goal: software success

Let’s start with the user*

We measure test quantity because it’s easy. Numbers feel safe

But it’s a proxy (and often a bad one) for reliability

Why? Because we’re disconnected from real usage, real risk, and real failure modes

*You know those pesky people we build software for in the first place

“When we lose connection to real-world behavior, we cling to what we can control: Code coverage. Test counts. Green bars. Metrics that look like progress, but don’t guarantee safety.”

What most teams actually do

That won't get better with AI –
the agent that just wrote your code
wants the tests to pass –
so it will write it that way –
rather than testing something meaningful

Habit, not intent	"Testing" that isn't testing	Misaligned goals	Why this happens	The real problem

• Most testing happens by gut feeling
• We write what feels right. What feels enough
• Developers often write tests for the code they just wrote, not for the system's real behavior
	• Bullshit-Tests
	  • Unit tests that mock everything except the actual behavior
	  • stubs that assume the happy path
	  • assertions that test for the existence of a response rather than its correctness
		• We often test for what's easy to simulate, not what's most likely to go wrong	• Devs are trained on tools, not test thinking
			• We learn to write unit tests, not to design tests
				• We're not testing what users actually do. We're testing what we hope they'll do

Test-after thinking: "I wrote this, I should test it," not "What does this need to survive?"	**Cargo cult testing: writing tests that look good, but don't build confidence**	**We don't model behavior; we only check mechanics**	**Testing becomes checklist work, not craft**	**This isn't just ineffective; it's dangerous!**

Don’t throw code over the fence and hope it works 

Coverage is a signal, not a goal

What you test in your fantasy world

Linear flows
no back buttons, retries, or dirty state

No constraints
infinite bandwidth, zero latency, perfect APIs

No edge cases
just one type of user, doing one thing, correctly

Everything mocked:
nothing ever fails, because nothing is real

Always perfect inputs:
valid data, right format, happy path

Magical. Elegant. Sparkly. Totally detached from reality

What your system actually looks like

Mismatched data
nulls, special characters, malformed fields

Stateful weirdness
users halfway through a process, dirty session storage

Network pain
flaky APIs, retries, 404s

Real integration
real payloads, real contracts, real dependencies

Time pressure
edge-of-month logic, timing issues, concurrency

Solid. Messy. Sometimes stubborn. But real, and it gets the work done.

There is a gap between your test suite and your system’s true risk landscape!

What testing should actually do

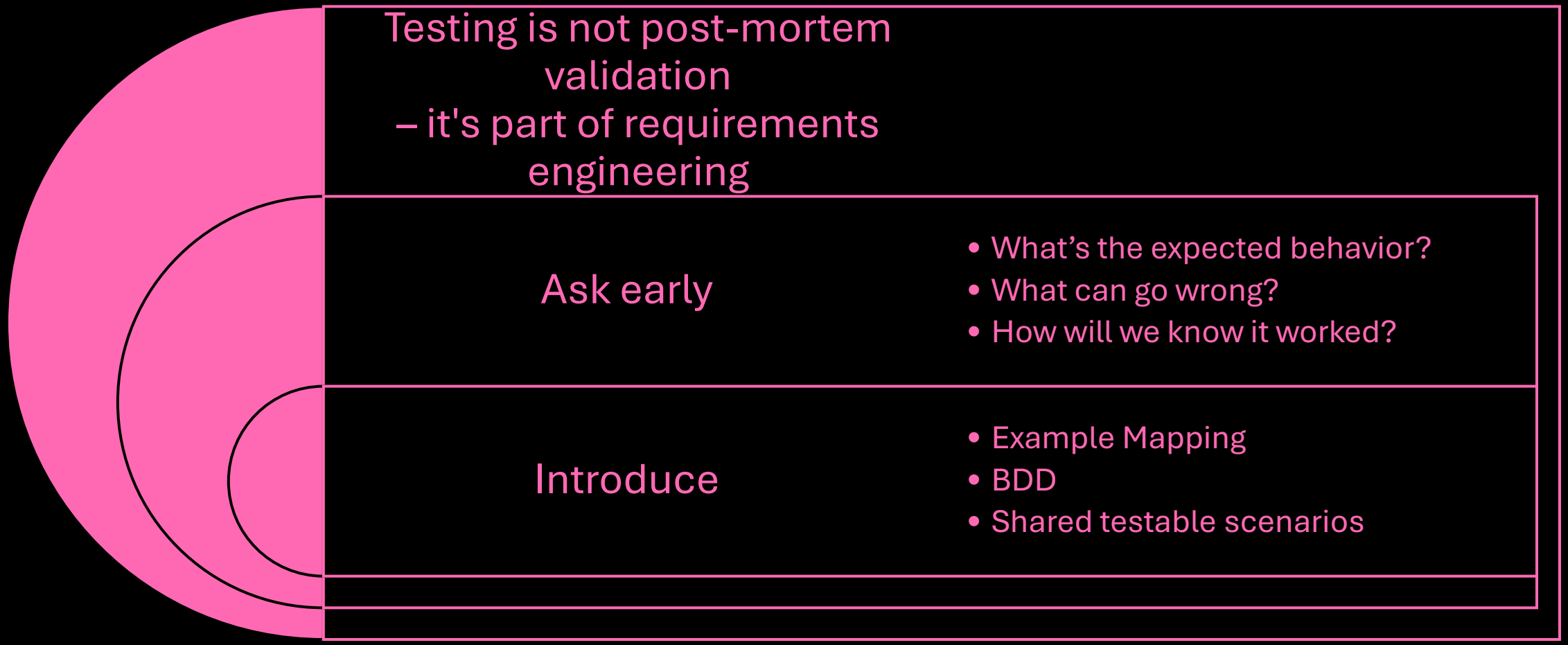
Unlocks speed	Improves design	Clarifies intent	Unites teams
no fear in releasing	bad code is hard to test	tests show what matters	common language: expected behavior

“You don’t write better software in spite of tests. You write it because of them.”

Shift left.

(yes, again)

If Testing happens after coding, it’s already too late



What’s the behavior we actually care about?

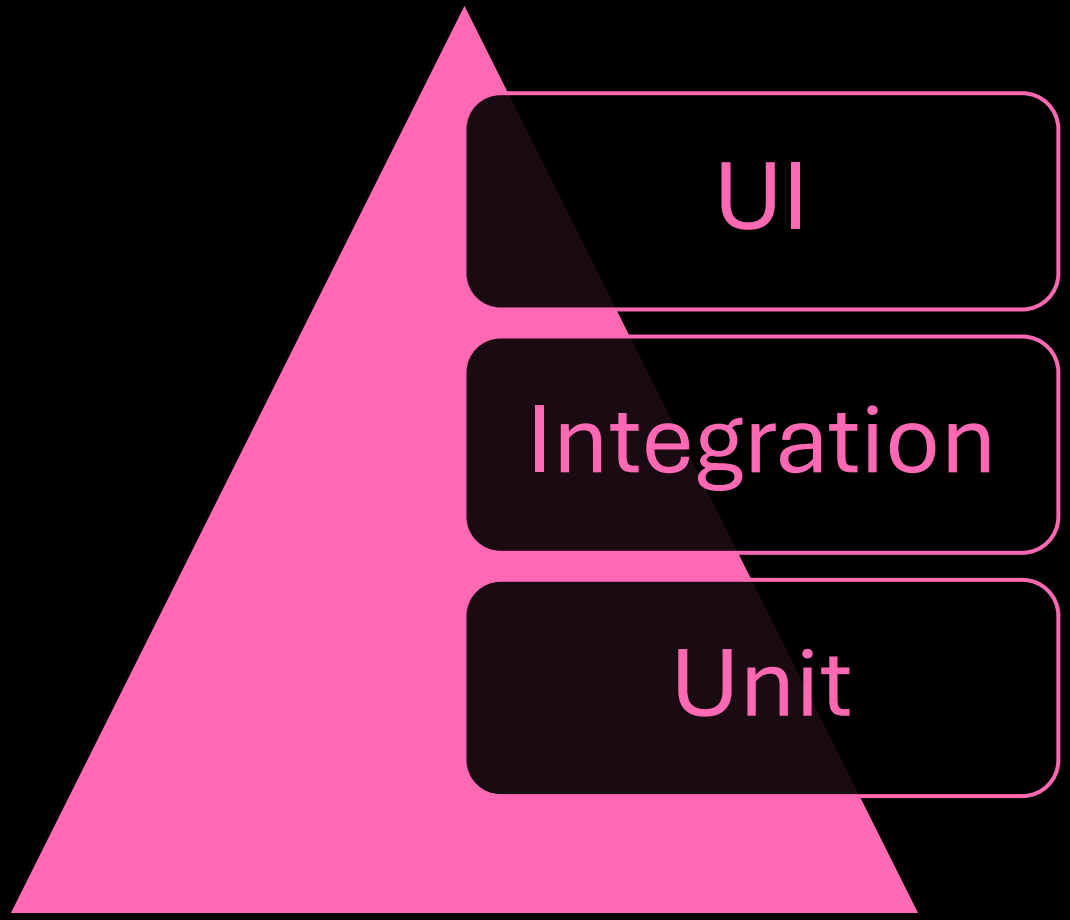
Behavior is what users see. Code is just the plumbing

Don’t test “that a method returns a list”

Test: “that customers with overdue accounts can’t book a ride”

Let’s challenge a sacred cow

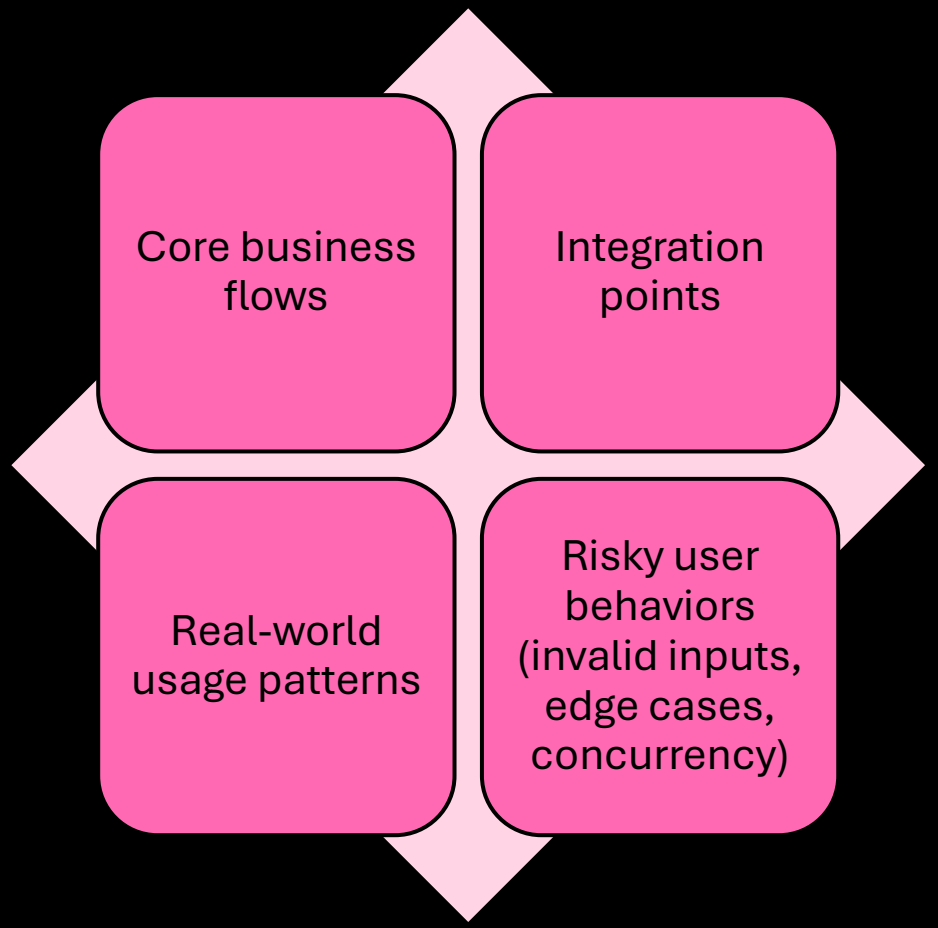
The testing pyramid



We’ve unit tested ourselves into a false sense of security

- The pyramid told us: test more at unit level, fewer at integration/UI
- It assumes low-level tests are fast, stable, and valuable. Sometimes that’s true, sometimes it’s busywork
- Problem: Most bugs aren’t in isolated logic; they live in flows, boundaries, contracts, data

Prioritize tests that catch real risk



💡 Ignore trivial logic.
If a test doesn’t reduce risk, delete it.

You don’t need 1.000 tests.

You need the *right* 42.

#Testgoals

- Coverage of behavior, not lines of code
- Fewer tests. Higher return
- Measuring assurance, not activity

Test design starts with variation

Test design is not just “write a test case”

Ask at a minimum

These aren’t edge techniques; this is how to test the right way

What are the valid/invalid inputs?

Which boundaries do apply?

What states can the system be in?

What transitions can occur?

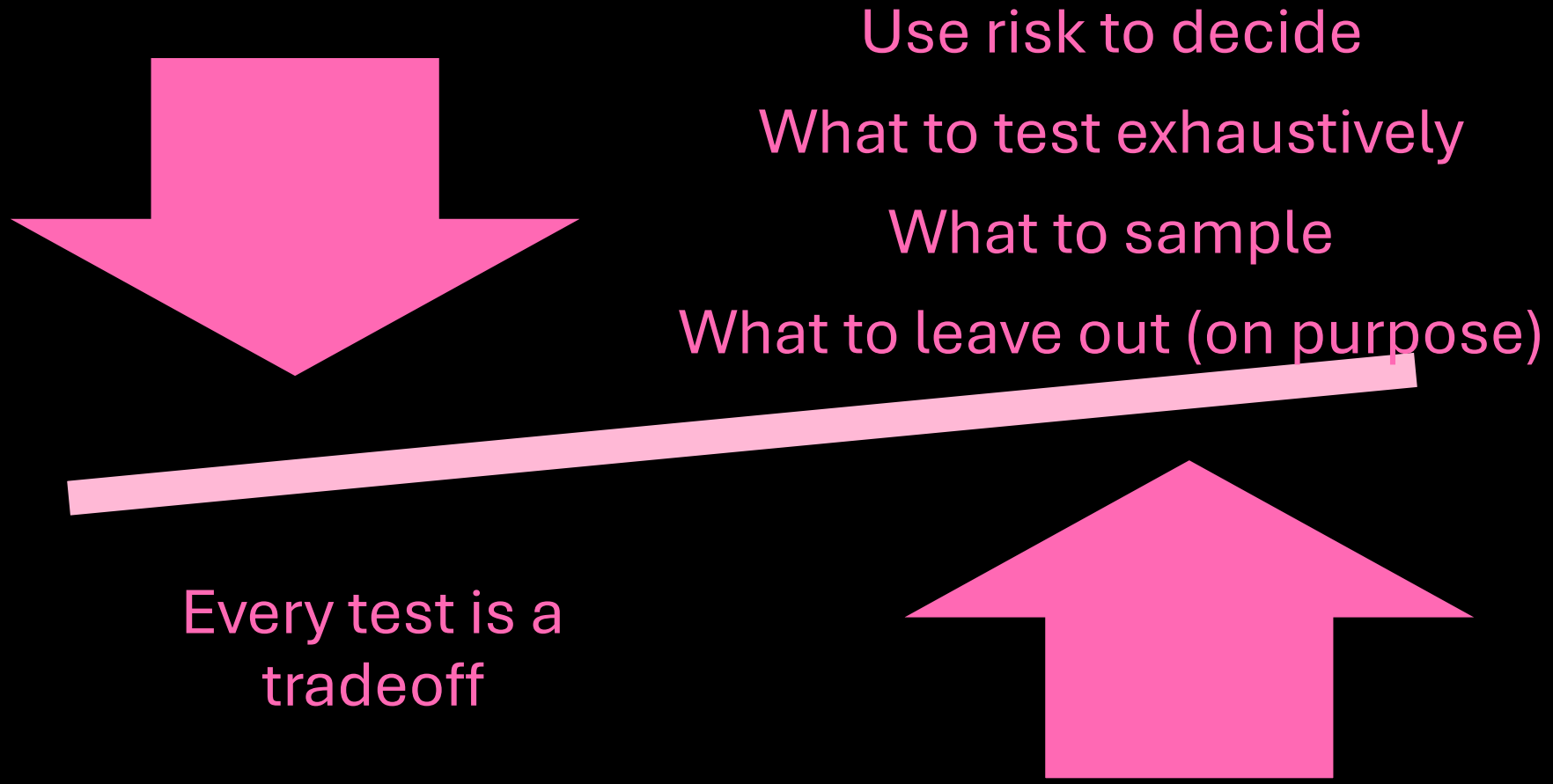
Equivalence partitioning

Boundary value analysis

state transitions

decision tables

Real maturity = confidence in what you didn’t test, because it didn’t matter



Good tests align teams, not just code

Shared examples align expectations

BDD and example mapping aren’t buzzwords; they’re communication tools

When tests describe behavior

- QA tests less manually
- Product owns more quality
- Devs stop guessing “what done means”

Behavior-Driven Development isn’t about Cucumber

BDD is a collaboration practice, not a testing framework

Its power lies in creating shared understanding between Dev, QA, and Product

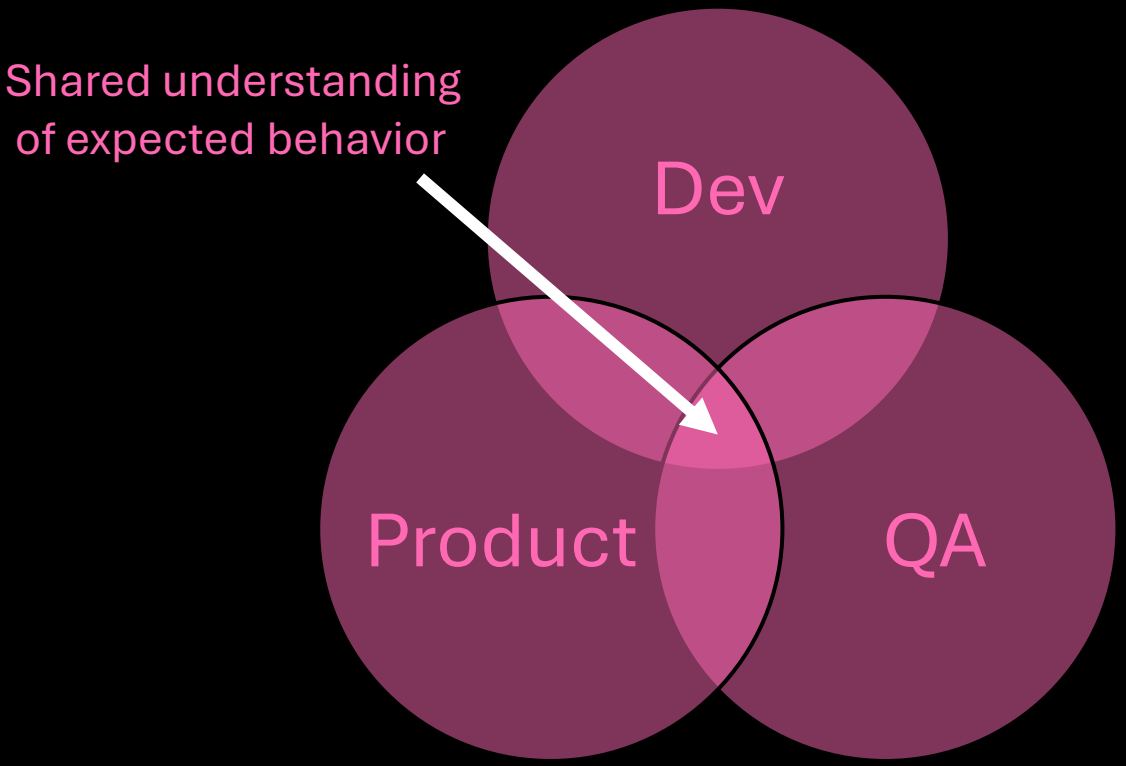
It forces teams to define:

- What a feature should do
- What counts as success or failure
- What edge cases must be handled

“Software Development is a Teamsport”

Software Development is a people business!

BDD helps you build the right thing, not just build the thing right



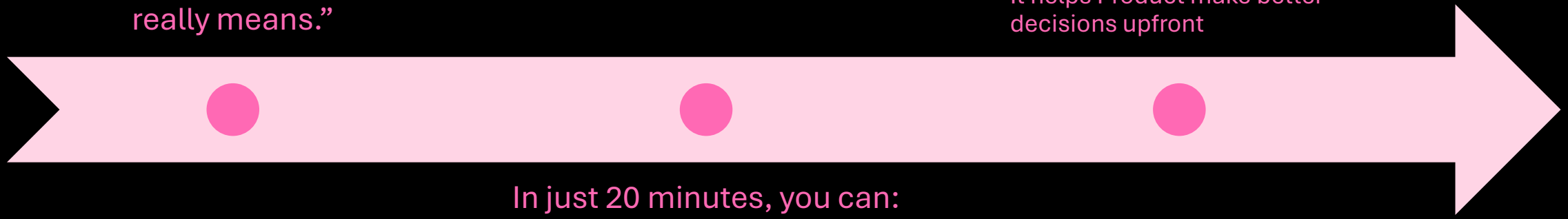
- BDD = Better conversations
  - Teams talk about what the system should do in specific situations
  - This reduces ambiguity before it becomes bugs
- BDD = Concrete examples
  - Instead of vague stories (“As a user, I want…”), we define real scenarios:
  - “If a user has no active subscription, and they log in, then they should be redirected to the upgrade screen”
- BDD = Aligning dev, QA, and product
  - Everyone agrees on the behavior we expect
  - These behaviors become the foundation for test cases, acceptance criteria, and implementation

“If your team is writing Gherkin scenarios that no one outside Dev ever reads; you’re not doing BDD.

You’re doing ‘Cucumber-Driven Development.’ That’s not the same.”

Turn vague requirements into testable scenarios

Example Mapping helps you go from “let’s build this feature” to “here’s what this really means.”



In just 20 minutes, you can:

- Capture business rules
- Brainstorm examples
- Identify ambiguities
- Write better user stories

Why it works:

- It prevents “late discovery” bugs (“Oh… we didn’t think of that”)
- It gives QA real scenarios before code even exists
- It helps Product make better decisions upfront

The best test you’ll ever write is the one that prevented a bad requirement from getting to devs in the first place

Turn user stories into contracts

What it is	Format Example	Why it matters
• Shared testable scenarios = the glue between teams • These should live *before* and *outside* the code • Everyone agrees upfront: “If this passes, we’re done.”	• “If a customer cancels before their trial ends, they shouldn’t be charged.” • “If an API returns malformed data, we should show a fallback UI.”	• Reduces finger-pointing • Prevents “that’s not what I meant” bugs • Turns quality into a shared outcome, not a job description

Testing can’t save you from blame culture

A lot of testing strategies aren’t about improving software. They’re about avoiding blame

We write shallow unit tests not to validate behavior, but to say ‘Look, I covered it.’

We comment “covered by test XYZ” to silence questions, not to build confidence.

QA runs the same regression suite weekly. No one knows what it protects. But hey, it’s documented

Product signs off on requirements they never really read. “If it’s not in the spec, it’s not my fault.”

Don’t try to automate culture

you can’t fix a cultural problem with more tech

You can buy the best testing framework

Add all the test coverage dashboards

Plug in AI code gen for test cases

But if your teams don’t trust each other, if requirements are vague, if blame is the operating model: all that tooling just automates the wrong behaviors

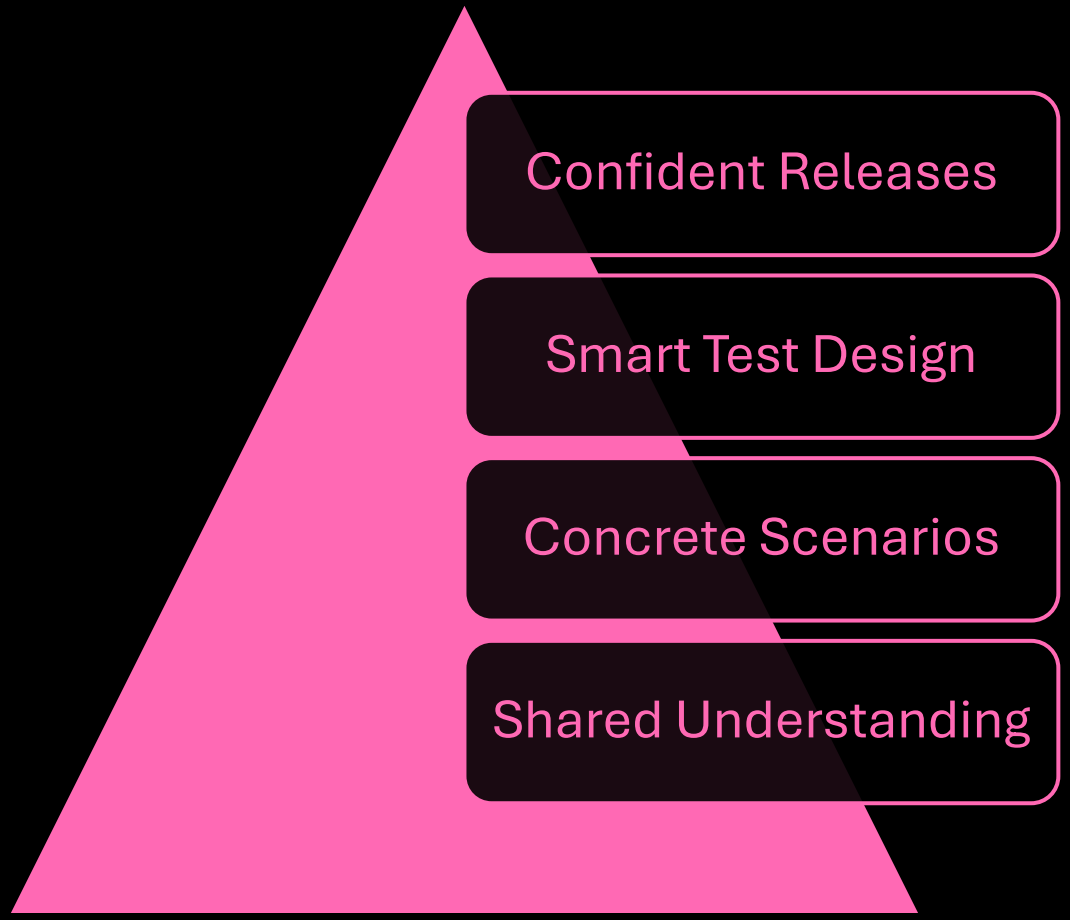
“If your test suite is just a legal defense strategy, it’s already too late.”

⚡ Without clear requirements

- Developers guess
- QA reverse-engineers intent
- Product rewrites expectations after bugs surface

BDD, example mapping, and testable scenarios all sit on the same base: requirements that make sense.

Requirements Engineering Is the Foundation of Testing That Works



Requirements engineering isn’t optional. It’s the prerequisite for software that doesn’t suck.

Real fixes for cultural problems

Blame avoidance	Shared responsibility for outcomes
Shallow specs	Example mapping & BDD
Testing-as-a-stage	Requirements + behavior-first mindset
Tool worship	Systemic thinking & team alignment

Homework – or what to do next Monday

Delete one test	Ask better questions	Model a scenario before you code	Call out a CYA pattern	Share one example before implementation
• ...that gives you no confidence and exists only to make a coverage metric happy	• “What does success look like?” • “What’s the riskiest part of this?” • “How will we know it works?”	• Try writing just one behavior in Gherkin or free-form: • “If [condition], and [state], then [expected outcome].”	• Spot it in a standup • Name it • Kill it with kindness 💖 • (“Are we testing this because it’s risky, or because we don’t want to be blamed?”)	• Not a test case • A real-world user story that you can validate together

Better testing isn’t about writing more tests.
It’s about making better decisions; together.

~Luise

> Session runtime complete.

> Running post-session diagnostics...

Knowledge transfer confirmed

Misconceptions challenged

Cultural assumptions flagged

Practical insights delivered

> Final diagnostic module: TESTING_STRATEGY

Coverage: Understood

Confidence: Redefined

Conclusion:

"Coverage makes you feel safe.

Confidence means you are safe.

Choose wisely."

> System state: Improved

> Readiness level: ↑ Elevated

> Shutting down LUISE.EXE...

Thank you for attending Codemotion Milan 2025!

{codemotion} CONFERENCE MILAN 2025

Don't forget to rate the talk!

Want the slides?

- Connect with me on LinkedIn
- Send a message “hot pink”

Linkedin.com/in/LuiseFreese

In case…

- You want to convince your boss
- Or your peers
- Or want to deep-dive into the methods with hands-on-workshops:
- Hit me up, I do this (and more) for a living ☺