

It's Probably You (Sample)

*Field Notes on Checking What You Believe
as an Engineer*

ILHWAN SEO

IT'S PROBABLY YOU

Field Notes on Checking What You Believe as an Engineer

Copyright © 2026 Ilhwan Seo

All rights reserved. No part of this book may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

This is an expanded English edition of *Growth Developer*, first published in Korean in 2019. It includes selected essays from the author's later writing, revised and translated for this edition.

The views expressed are the author's own. Accounts of professional work have been generalized to remove confidential and identifying details. Invented scenes and the reconstructed logistics design are identified in the text and in "How to Read This Book."

All trademarks, product names, and company names mentioned in this book are the property of their respective owners and are used for identification purposes only; their use does not imply endorsement.

Second English edition, 2026.

Paperback ISBN: 9798178778951

Cover and interior design by the author.

Contents

<i>Preface to the Revised Edition</i>	4
<i>How to Read This Book</i>	6
PART 1 · START BY CHECKING WHAT YOU BELIEVE	9
1 It's Probably You	10
2 A Name Does Not Make It So	12
3 From Reproduction to Cause	18
4 Before You Trust the Crash Report	21

Preface to the Revised Edition

I wrote this book for engineers who get stuck on judgment more often than on technology: from the first years on the job to the first time leading a team, anyone who has stopped asking questions when told “we’re agile,” “that’s just candid feedback,” or “the CEO wants it.” Product managers and managers who work with engineers will recognize their own meeting rooms in Parts 4 and 5.

Most chapters start from something that happened to me; scenes invented to explain a point are identified as such. A crash collector taking twelve million app loads a day. A server that passed its load test and still leaked a gigabyte an hour. Diagnostic data locked inside an instrument. A proposal that stalled on its way up five layers of management. I have tried to write down what I believed, which constraints shaped the choice, and how much of the outcome I verified. I hope the questions behind the conclusions stay with you longer than the conclusions themselves.

The “you” in the title includes me. It is not an attempt to make individuals answer for every failure. Start with what you can examine and change while also looking at organizational responsibility and structural limits. You will need to test whether these experiences apply to your situation.

The book began in 2019 as *Growth Developer*, a collection of things I wanted to tell junior engineers. Since then I have worked in games, diagnostics, e-commerce, and recruitment, and returned from leading an organization to building products myself. Each change tested the reach of familiar judgments. For this 2026 revision I combined chapters that repeated the same advice, moved the events to the front, and reorganized the book into seven parts and thirty-two chapters. The new chapter “A Name Does Not Make It So” asks a question of the whole book. I too have used respected

names for technologies and methods to reach premature conclusions.

Ilhwan Seo

How to Read This Book

You can read in order or start with the problem closest to your work. The first two chapters offer a basis for the rest: examine your own assumptions, and look beyond names to purposes and results. I hope you apply that same scrutiny to the technical and organizational judgments that follow.

If you are unsure where to begin, try one of these routes.

- If you are early in your engineering career, start with chapter 3 on investigating a failure, chapter 8 on recognizing repeated work, and chapter 20 on estimating a schedule. Then explore Parts 1, 2, and 4. Chapter 6 follows a C++ memory investigation; you can return to it when object lifetimes become relevant to your work.
- If you work with engineers but do not write code, start with chapter 18 on helping people participate, chapter 20 on schedules, and chapter 24 on relaying decisions to a team. Parts 4 and 5 develop these organizational questions. You do not need to read the technical investigations first.
- If you have just become a lead, start with chapter 7 on incident measures, chapter 22 on creating growth opportunities, and chapter 26 on performance reviews. Continue with the rest of Part 5. Appendix C offers questions for a team discussion; choose one that relates to work you are doing now.

A note on the cases. The numbers belong to the systems and measurement conditions of the time; they do not promise the same outcome from the same tools. Dialogue and scenes invented for explanation are identified as such in the text. The logistics design in chapter 17 is a reconstruction combining real experience with

general design principles. The 2021 view of AI is kept as the judgment of its time and set alongside later experience.

If a technical detail is unfamiliar, first follow what failed and which decision changed. Return to the details when you want to examine the evidence. Rather than copying a method, ask which conditions differ in your own work. The appendices contain checklists drawn from the chapters, discussion questions for each part, and the books referred to.

PART 1

Start by Checking What You Believe

*Compare assumptions and measurements
with what actually happened. Start here if
you have ever blamed the library first.*

It's Probably You

A colleague built an online game server using `boost.asio`, a C++ library for network communication. The server kept crashing. Near the top of the stack trace were calls that appeared to belong to the library. The developers concluded that the library was at fault and began looking there.

The fault was in the game-server code: it was not using the library as intended. The stack trace showed where execution had failed. It did not establish who had caused the failure. We had filled that gap with an explanation that excluded our own code as a possible cause.

This does not mean external libraries are free of bugs. Starting with our own work is a practical choice, not an exercise in self-blame. We can inspect the code we changed and the assumptions we made, and act on what we find. Before assigning the fault to something harder to investigate, we should ask what evidence we have gathered closer to home.

Sometimes we write a long explanation of why a failure was not our fault but leave no useful instructions for reproducing it. Accountability matters. During an outage, however, the immediate job is to gather what we need to restore the service. When self-defense takes the place of investigation, users wait longer.

Reporting a problem and responding to it

Consider an illustrative example. Someone posts, “The query API is failing. Can anyone investigate?” Nobody knows who owns it, so everyone waits. Silence alone does not prove indifference:

ownership may be unclear, or an investigation may already be underway without being communicated. Even so, we can inspect what we know or tell the reporter where the request has been directed.

The reporter also needs to give colleagues something they can investigate. “The socket disconnects” is less useful than the environment, timing, and request that preceded it. An executable command, a successful input alongside a failing one, and a description of recent changes give everyone a common starting point.

If my code caused the problem, I need to say so and fix it. If a colleague’s code caused it, I need to explain what fails and under which conditions. Finding a bug should not turn a conversation into a contest. It should give us a way to prevent the failure from happening again.

The “you” in this book’s title includes me. Not every problem is an individual’s fault; organizational and technical failures deserve investigation too. The point is to avoid exempting ourselves before that investigation has begun. That small change alters what we do next.

A Name Does Not Make It So

I once wanted to put a work discussion into an email. I was told that our organization used chat instead. That was our culture. I was not trying to establish whether email was universally better than messaging. I wanted several people to be able to read a complete account and find the reasoning again later. Yet the discussion shifted from how the information should travel to whether I was following our culture.

The word culture can save a great deal of explanation. Shared practices are useful; we cannot renegotiate everything from scratch. But something changes when we cannot ask what a practice is for. A method intended to help us do a job becomes a reason to prevent us from doing it differently. The label stops supporting thought and starts shutting it down.

I have done this myself. Early in my career, NoSQL, a broad label for nonrelational databases, and the CAP theorem were fashionable. CAP describes a limit during a network partition: a system cannot guarantee both that every read reflects the latest write and that every functioning node answers requests. Without understanding them well enough, I told colleagues that not using Cassandra was foolish, or that a service like ours needed MongoDB. My confidence came less from examining our data and failure conditions than from knowing the names of new technologies.

The embarrassment was not simply that I had picked the wrong technology. Those tools could have been appropriate in some circumstances. The problem was that I had reached a conclusion before knowing what I needed to establish to justify it. I learned the vocabulary faster than the judgment. What I had was a position to defend, not a decision I could explain.

Meeting every day is not enough

The following Scrum scene is an illustrative example, not an account of a particular workplace. A team meets at the same time every morning. People describe yesterday's work, today's plans, and anything blocking them. Attendance is consistent and the meeting ends on time. But the same obstacle comes up for days. Decisions remain unmade, and the plan stays unchanged even when the goal is clearly in trouble.

The team has faithfully held its meeting. That alone does not establish that it responds well to change. The Daily Scrum is intended to inspect progress toward the Sprint Goal and adjust upcoming work. Its format supports that inspection; following the format does not substitute for inspection and adaptation.

Canceling the meeting would not necessarily help. The team may lack authority to act on what it discovers. Nobody may own the coordination needed to get another team's answer. Shortening the meeting could merely reduce opportunities to make those problems visible. The useful question is what can change afterward: who will act, when an answer is expected, and whether new information can alter today's work.

The same test applies to declaring a team agile. Does the work actually involve examining useful results in small increments and applying what was learned to the next decision? Blaming people whenever a plan changes makes adaptation difficult, whatever the meeting is called. Constantly abandoning work in response to new requests is not enough either. We need to explain why a change is warranted and what it will cost.

Candor should help the person hearing it

Not long after I joined one company, a colleague I was close to came to warn me that they had heard negative things about me.

Apparently, I talked too much. I was stunned. I was still new, and I didn't think I had been speaking much at all.

What was I supposed to change? There was no example of what I had said or where I had said it. Had I interrupted someone, explained something at too much length, or spoken when I did not need to? I had no way to examine what I might have done wrong or decide what to do differently. I came away with no useful guidance, only hurt feelings.

I told my colleague:

“Please don’t pass along comments like that again. They don’t help me. And please tell the person who gave that feedback that I’m willing to hear it directly from them, with specific examples.”

I was not refusing criticism. I was asking for something I could examine. My memory might be incomplete; someone could have been troubled by behavior I had not noticed. If so, we needed a way to understand that difference. But an unsupported “That’s what people are saying” leaves me guessing who dislikes me instead of working on my behavior. What reached me that day was gossip, not feedback.

If we want to help someone, that intention needs to show in what we give them. Separate what we observed from how we interpreted it, and explain its effect on working together. Listen to their account and leave room to revise our interpretation. That conversation gives the other person room to consider the feedback. We cannot demand

acceptance of a verdict without examples and then blame the recipient for being unreceptive.

Sometimes a person cannot comfortably raise a concern directly and needs someone else to convey it. Even then, the recipient needs enough specific information to improve and an opportunity to explain their account. Indirect feedback is not automatically gossip. The problem is passing along a judgment the recipient cannot examine, without explaining what could change.

Calling such remarks Radical Candor would not change their substance. Radical Candor combines personal care with direct challenge. Care is not a technique for surrounding criticism with pleasant remarks. It involves offering usable information, listening for what we may have misunderstood, and being willing to support another attempt. Candor is not an opportunity to demonstrate how bold or enlightened the speaker is.

Consider an invented design review to make the difference concrete. “You didn’t even think of this? You lack the basics” judges someone’s ability without explaining the problem. We could instead say: “If payment succeeds but the response is lost, a retry could cause the payment to be processed again. If there is already protection against that, please show me what I missed. Otherwise, let’s agree on how to identify the same request and how to test retries.” This version offers a condition, a consequence, a question, and a next action.

The recipient may still feel uncomfortable. Being challenged on an important issue can be difficult. Discomfort alone does not prove that the feedback was wrong, just as good intentions do not prove that its delivery was right. We should examine what information we gave, whether we heard the other person’s account, and whether we opened a workable path to improvement.

The recipient also has responsibilities: investigate what needs checking and explain disagreements with evidence. But we should not assume that two people with different ranks and control over evaluations are equally free to speak. If a manager invites

disagreement but dissent repeatedly carries a penalty, people learn from the consequences rather than the invitation. Candor cannot be established by declaration alone.

The purpose of feedback is to help people change how they think and act so they can work better together. If we mean that, our interest must continue after we have spoken. Check what changed, explain differently when needed, and admit when our criticism was mistaken. Immediate change is not the test of success; improvement can take time and support. But leaving hurt and distrust without a path to improvement cannot be justified by calling it feedback.

The decisions behind a role's name

Working across teams, I have encountered PMs (project or product managers), POs (product owners), and PLs (project or part leaders).¹ People can attach different responsibilities to the same title. A role needs agreement on who sets priorities, resolves schedule conflicts, and makes the final decision. Chapter 25 describes my effort to take on a larger role. Chapter 26 examines an evaluation where a superior's impression competed with documented evidence despite a rigorous process.

This is not a proposal to abolish titles or procedures. Evaluating people from memory or renegotiating every role has costs too. Look for work that falls outside the assigned roles and evidence that the process collects but never uses.

Explain it without the name

When "it's our culture" ends the explanation, a question about a method can become a judgment about a person. Someone questioning a meeting is called resistant to change; someone asking for examples is called unwilling to accept feedback. Rather than

guess at motives, examine which questions were allowed and what changed afterward. Apply the same test when someone challenges a method we like.

I find it useful to try explaining the choice without its name. In that opening discussion about chat, I did not need to establish an “email culture.” People needed a complete account they could read together and reasoning they could find later. If chat could provide that, we could use it. If it could not, we needed another approach.

A bothersome procedure may still prevent a problem. Before removing it, check what it protects and who would bear the cost of changing it. A concept becomes useful in my work when I can explain the purpose and choice without its label, then revise the choice in response to results.

For the concepts discussed here, see the [Scrum Guide](#), the [principles behind the Agile Manifesto](#), and the [official explanation of Radical Candor](#). The workplace examples and discussion of judgment examine questions that arise when applying those ideas.

-
1. PM is a project or product manager and PO a product owner. PL is a title used mainly in Korean and Japanese organizations, close to a project or part leader. [↩](#)

From Reproduction to Cause

My team once built an automation program in which users wrote scripts in a front end and a backend engine executed them. Before execution, regular expressions replaced certain patterns. One particular input caused catastrophic backtracking, making the engine appear to hang. A capable engineer on the team correctly located the problem in the regular-expression code and immediately started tuning it. After considerable trial and error, he reported a fix without having found the input that triggered the failure. The problem returned after deployment. Only then did he search user-action logs and the database for that input. Reproducing it made a properly verified fix possible.

Locating the faulty code did not tell us which input triggered the failure or why. Before changing it, we needed an input that reproduced the failure. After the change, we needed to run that input again and test nearby conditions.

Some bug reports contain no reproduction steps, or describe failures too rare to reproduce readily. That makes investigation difficult, and such issues can remain unresolved for a long time. It is tempting to blame the reporter for incomplete information, but reporting the problem has helped us. As the first chapter argued, the useful response is to investigate the software and work with the reporter to recover the missing conditions.

Reproducing a problem means tracking down the state or action that caused it and deriving a comprehensive set of test cases from it. It also means writing code compactly and modularly so that each part can be tested on its own. Repeating that process teaches how to write and maintain robust software.

Reduce the cost of checking

Finding a way to reproduce a problem matters, but so does making the reproduction itself *simple*. In a system where many processes cooperate to carry out one business operation, finding and fixing problems is rarely easy. When one operation is handled jointly by several instances across the network, reproducing a symptom can require running every related program. In such cases it is often far more efficient to isolate the essential functionality and run tests against that.

Suppose editing the code for a fix costs 1, and running the fixed code to confirm it works costs about 4. If the bug is fixed quickly, nothing is lost. But the longer the problem drags on, the more the time spent starting up and verifying outweighs the time spent on the code itself. When that time becomes a burden, the reproduction environment is worth improving. For example, spend 10 to build the environment and run it once, and verify for a cost of 2 from then on.

Compare the cumulative cost of repeated verification. This is an illustrative example. The cost of editing the code is the same in both environments, so it is excluded. The first-round cost of 10 for the fast setup includes building the environment and the first run.

Round	Old setup: total cost	Fast setup: total cost
Round 1	4	10
Round 2	8	12
Round 3	12	14
Round 4	16	16
Round 5	20	18
Round 6	24	20

Once a fast way to reproduce the problem is in place, it pays off from the fifth round, and the gain keeps accumulating afterward.

Choose the solution from the problem's constraints

Reproduction is more than seeing the symptom again. It lets us test which changes alter the outcome and compare possible solutions. A core capability that must run for years deserves a different investment from a one-use tool. We also need to decide the relative importance of response time, security, reliability, and scale.

Wanting to try a new technology can be a useful starting point. Committing to it requires an explanation of how it meets the constraints at hand. If the choice later proves wrong, a record of our reasoning helps reveal what we misunderstood. Careful thought does not excuse a poor outcome; it makes the decision available for examination.

Keep the reproducing input and environment alongside the reason for the change and the verification results. They give the next developer a starting point. A useful fix does more than work once: it makes clear what was checked.

Before You Trust the Crash Report

Around 2016, we were paying an external service to collect crashes from our mobile games. Buying an existing product was a reasonable choice, but in operation it did not capture enough of the information our games needed.

A crash record from native game code could be as large as 689 KB. That minidump contained information needed for diagnosis. With the service we were using, however, uploads could be limited, truncated, or lost. When a failure happened, we could be missing the evidence required to investigate it.

So we decided to build our own. Developed from 2016 through 2020, the system collected crashes, app-load events, and network logs across Android, iOS, Unity, and Unreal. It symbolicated memory addresses into function names and source locations, aggregated the results, and displayed them on a live dashboard. The purpose was to meet game-specific needs the general-purpose tool had not met.

Building it also made our team responsible for missing records and failures. To the people using an internal tool, a well-named architecture mattered less than whether they could trust its output.

What the first major launch exposed

The first large MMORPG to rely seriously on us put the system to the test. It saw about twelve million app loads a day and a crash rate of 8–12 percent. Records arrived in bursts that our collection pipeline had to absorb.

Kafka stores and delivers streams of events. Our consumer program read its messages asynchronously, accumulated them in memory, and let workers process them. Whenever arrivals outpaced processing, the backlog grew. Eventually, the collector ran out of memory and stopped. Individual jobs were not unusually difficult; we were accepting more than we could process.

The key was backpressure: regulating intake to match processing capacity. We introduced synchronous flow control, processing one batch before fetching the next. Sustaining a manageable rate mattered more than fetching as fast as possible. After the improvement, we recorded throughput of about 16,000 requests per second in production. The observation window and instance count are not preserved in this account, so this is not a benchmark for another system.

We also kept original events immutable and treated the dashboard as a result that could be recomputed from them. If processing failed and the original remained available, a reprocessing queue could recover it. Unique event identifiers prevented duplicate aggregation.

This did not recover every event. I recorded a loss rate at the time, but this account does not preserve the two observation points or denominator. I cannot use that figure to quantify loss across the reporting path. Preserving originals and supporting reprocessing still left gaps. About 464 GB of original records accumulated over six weeks, providing a basis for rebuilding aggregates.

We improved how quickly developers received results too. Symbolicating one crash originally took 20–30 seconds. Three rounds of improvement reduced that to 0.45 seconds, about 44–67 times faster. Developers could investigate within the flow of discovering a problem rather than wait to return to a report later.

Other teams began asking for it

Handling that first major launch helped establish internal trust. Another studio approached us, and early users introduced the system to other teams. Adoption grew from a few games to more than fifty, and eventually to over two hundred across the platform.

An explanation alone rarely earns trust in infrastructure. It has to help during an important release. Supporting the first team and resolving the problems exposed by its use gave the next team a reason to adopt it.

Observe the reporting system too

A low crash count does not by itself establish that a service is healthy. There may be fewer crashes, or the reporting pipeline may be obstructed. That is why we measured loss separately. Before interpreting a graph, we need to understand how its numbers are produced.

What are we calling missing?

The following is an illustrative example, not a reconstruction of our measurement. Suppose an intake log records 100 distinct event IDs accepted from 10:00 up to, but not including, 10:01. We expect all 100 in the final store by 10:06. The intake log must remain available independently of the output being checked.

At the deadline, the final store contains 100 rows from that group. That sounds complete, but two rows are duplicates: only 98 distinct IDs arrived. Comparing IDs reveals the two still missing. One of them arrives at 10:08.

Observation time	Stored rows / distinct IDs	Accepted IDs not yet found
10:06 deadline	100 / 98	2 of 100
10:08 check	101 / 99	1 of 100

At 10:06, 2 percent had missed the deadline. Calling that permanent loss would have been premature: one was late. The other still needs investigation. Counting rows alone would have hidden both, and later would have made delivery look greater than 100 percent. Keep the accepted group fixed while checking what arrived.¹

This comparison covers only the path from acceptance to the final store. It says nothing about a crash that never reached intake. If the intake log is also incomplete, even the denominator is unreliable. Name both observation points and the deadline before calling a gap “loss.”

Learning a service’s ordinary behavior serves the same purpose. We should keep improving automated detection and recovery while learning the normal patterns of traffic, latency, and failure. When several alerts fire together, the sequence of changes may tell us more than any individual threshold.

Instinct is useful as a signal to investigate. If the dashboard is green but users are experiencing something unusual, check again. A bad feeling is not a reason to dismiss measurements either. Operational experience becomes valuable when it helps us compare the reporting path with actual user behavior and turn suspicion into evidence.

1. For a discussion of delivery timeliness and completeness as separate concerns, see the Spotify event-delivery case in Google’s [Site Reliability Workbook](#), “[Data Processing Pipelines](#)”. The event IDs, counts, and times above are invented for this example. ↩