No. 06
July 2026

testing experience

The Great Unbundling: Rethinking What It Means to Test

A Journey of a Strategic QE: Reinventing Yourself

Practical Principles for Agentic QA

One Change That Made Our QA Pain Hurt Less

Autoethnographic research made me look at testing from a new angle

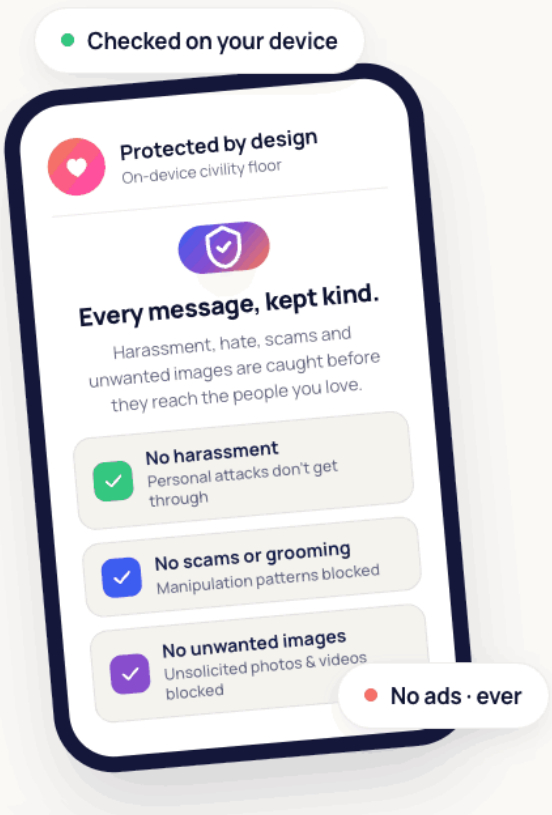
KindChat

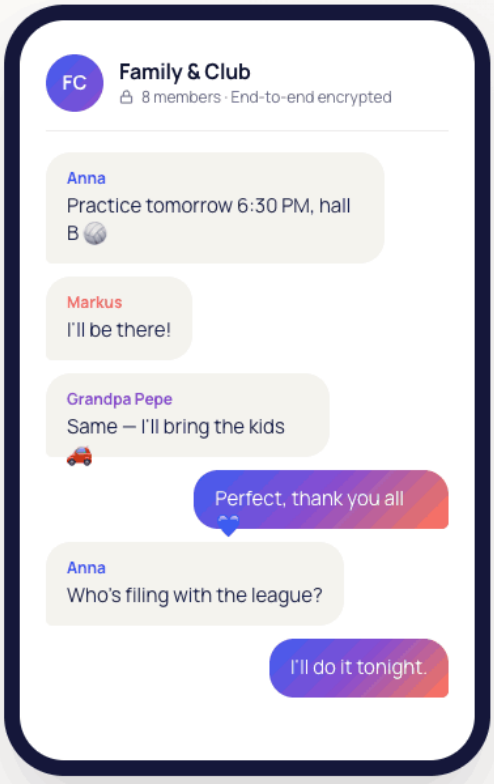
THE MESSENGER
MADE IN EUROPE

MESSENGER · PRIVACY BY DESIGN

Kind Chats.
Strong Conversations.

A privacy-first messenger built for respectful conversations — for **your family, your club, and the people you care about.**





End-to-end encrypted

Only you and your contacts can read your messages — never us.

Civility, built in

A civility floor checks each message right on your device.

Protection, by default

Safe for kids, parents and anyone who needs it — from day one.

FREE FOR EVERYONE, EVERY DAY

kindchat.app

Download free on iOS & Android · Plus from €2.99 / month

Download on the App Store

Get it on Google Play

Made in Europe · GDPR Compliant · End-to-End Encrypted

EDITORIAL

Dear all,

Summer has just kicked in, and across Europe the thermometer shot up to the sky. In Germany, all the warm-weather records became obsolete. We are at new highs.

But that is only for the weather. For those who love football, Germany was knocked out, while Spain is still fighting as I write these words. Let's see how it ends...

I want to thank all the authors who sent us an article. Without your work and dedication, the Testing Experience could not exist.

This time I wrote an article myself. I needed to express my thoughts about where our profession is heading. Normally I talk about it one to one, or in small groups, but this time I wanted to reach a wider audience.

I have spent the last months vibe coding. I built the entire back office of trendig for our trainings and conference. As a customer of our trainings and conference, you will experience it over the coming weeks. We are still adding a few features and testing intensively. We know you are coming for us ;-)

I also created a new product for the public: KindChat. KindChat is a messenger with a civility floor that checks each message right on your device. No harassment, no hate, no scam, and no unsolicited images or videos.

We are now in the beta phase and will go public on September 1st. We are testing a lot, because we want you to have a good experience. If you are at the Agile Testing Days, I can tell you how it started and how the process went. I learned a great deal, and I will share it.

Last but not least, I want to remind you that if you cannot make it to the Agile Testing Days in person, you can still attend online. You can get a free online ticket for the Agile Testing Days if you book, or have already booked, a training with trendig this year.

I hope you enjoy this issue of the Testing Experience and find plenty of hints and ideas to help you at work and to grow.

I wish you a wonderful summer, and I will see you at the Agile Testing Days at the latest,

Yours
Pepe Díaz

Impressum

EDITOR

trendig technology services GmbH
Kleiststraße 35
10787 Berlin
Germany

Phone: +49 (0)30 74 76 28-0
Fax: +49 (0)30 74 76 28-99

E-mail: hi@trendig.com
Website: www.trendig.com

EDITORIAL
José Díaz

LAYOUT & DESIGN
Sabrina Ungaro

WEBSITE
www.testingexperience.media

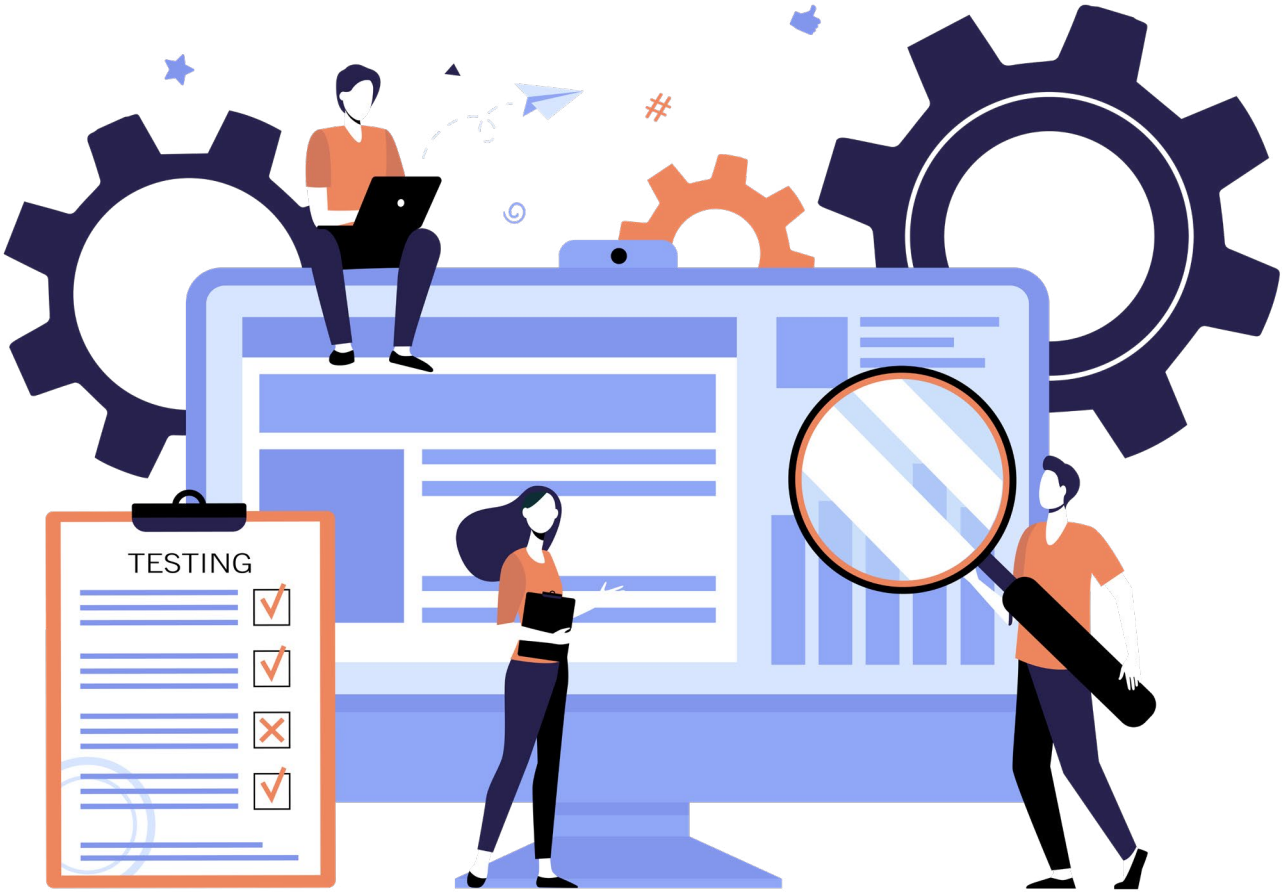
ARTICLES AUTHORS
editorial@testingexperience.media

ADVERTISEMENTS
editorial@testingexperience.media

PICTURE CREDITS

©All illustrations in the magazine are designed by Magnific
©Cover designed by Google Gemini (AI-generated)

CONTENT



Redefining the Smoke Test: The End of Manual

Author: Christian Baumann

> *SMOKE TEST*
> *A test suite that covers the main functionality of a component or system to determine whether it works properly before planned testing begins. Synonyms: sanity test, intake test, confidence test*
> *https://glossary.istqb.org/en_US/term/smoke-test*

When Smoke Was Manual

A smoke test suite in software development usually is a quite limited subset of the regression test suite. Its scope is to test the very main features of the application under test, to figure out if the application is testable, or if it's completely broken.

This is how I learned about smoke testing roughly 20 years back, in my first software testing job right after university. And it made sense back then. In a development organization with round about 25 teams, which were working on a huge monolith (a fat client with a central database). There were 5 test teams, with approximately 50 testers in total, being responsible for dozens of different modules in the application. There was even a dedicated test environment team with around 5 people, who did nothing but taking care of installing updates and making sure the various test environments were running smoothly.

Imagine a new version of the application being deployed to the test environment which needs thorough testing. All fifty testers starting up the application, which first meaning downloading a couple of gigabytes from the central server (still a bare metal server, standing in some server room on site), to get the latest version on their local computers. This already took quite some time, as computers were not that powerful in those days (e.g. SSDs were not a thing back then).

Imagine, the download and installation is done for every tester, the application is loaded, and they all notice after couple of minutes, that the application is totally broken. They all file a bug ticket, or maybe two, then when they realize that the version is malfunctioning and report to the test manager. Until now, at least half an hour has passed, and the test manager must now deal with 50 testers, all reaching out to him or her, all wanting to report that there is a major issue with the application's testability. Quite the waste of time, isn't it?

Because of that, we usually did smoke tests: After deployment of a new version in the test environment, a small group of testers (5 to 10) tested the main modules, and therein the most important flows. Just to figure out: Is the build completely broken? Is this version testable? It was a relatively quick thing: 30 - 60 minutes effort per tester. They then informed the test manager who took decision to whether or not to start a full regression test cycle with all 50 testers.

Shifting smoke to the machine

Also today, people are still asking for smoke testing a new version of the application under test, before the start of the "real" testing. Is that still appropriate? Times have changed, so have processes and technology.

Nowadays having batteries of automated tests is the standard. They run on different layers and with different purposes (unit, integration, e2e, etc.). Also, these tests run regularly in CI/CD pipelines, on commits and merges, as well as before deploying to test environments and production. Running these tests is cheap, they are there anyhow, and you can configure your pipeline to run them automatically. You can even run different subsets depending on the purpose. You don't even need to start them manually; they are automatically run when a certain trigger occurs. The pipeline even can spin up its very own test environment (frontend, backend, database and whatever other component is needed) and tears it down automatically when not needed anymore. "Infrastructure as code" is the keyword here, thanks to containerization and similar techniques.

Also these tests are fast: unit tests run in a few seconds, integration tests do take a bit longer, but still in a matter of seconds, and even automated UI tests are usually run within minutes, if yours take hours, they shouldn't! If so, let's talk. So within minutes you can have your application regression tested, and you will have an idea whether it is working or not.

Of course a reasonable amount of exploratory testing is needed on top, since automation cannot replace manual testing!

So, do we still really need to "smoke test" deployments before starting full testing? Considering having the application covered with a sufficient number of automated tests, I don't think smoke testing is still necessary today. At least not in the sense that we used the term in the past.

The core idea hasn't changed: find out quickly whether a build is worth testing thoroughly. But the

way we apply smoke testing today is very different from those early, manual checks. So, if the old, manual "is this build even testable?" ritual doesn't fit anymore, what does smoke testing look like in today's pipelines and environments?

POST-DEPLOY CANARY SMOKE CHECK

In the context of CI/CD (Continuous Integration/ Continuous Deployment), a "Canary Deployment" or "Canary Release" means pushing a new version of a software to only a tiny percentage of users, before rolling it out to the complete user base.

The term "canary" comes from a historical practice in mining: Miners carried canary birds with them, because those birds are highly sensitive to toxic gases. A sick or dying bird was an early warning sign to evacuate. In software, a small initial rollout acts as canary, to find out if the new code "is toxic".

A post-deploy canary smoke check is a tiny, automated suite that answers one narrow question: *Did the deployment succeed and is the instance usable?*

Right after rollout, it hits key endpoints for e.g. 200 (successful) responses, confirms the database and dependent services are reachable, and runs a single basic business flow; if this fails, the instance is rolled back before any real users or downstream testers rely on it.

PIPELINE SMOKE TEST / BUILD VERIFICATION

Today, a pipeline smoke test is a lean, fast subset of your regression checks. Only now, instead of a manual activity, it runs automatically on every single commit or pull request. It usually combines the quickest unit tests, a few high-signal integration checks, and maybe one happy-path end-to-end flow. The whole point is to answer one simple question: *is this change safe to integrate at all?*

This mechanism is basically an automated gatekeeper for your codebase. It ensures that fundamental stability is verified before a human ever even looks at the change. But to make this work, you have to be absolutely ruthless with what goes into this suite. You can't be checking edge cases or minor UI alignment issues here. Instead, you need to focus on the "walking skeleton" of the application.

Imagine it like this:

- Can the service actually start?
- Is the API schema still intact?
- Can a user perform the absolute primary action?

If these core assumptions aren't met, the remaining hundreds of tests are completely irrelevant!

Moving this check right into the pipeline fundamentally shifts the responsibility for basic stability back to the developer. Imagine a developer pushes a broken commit. The smoke test fails, the pipeline halts right there, and the fix has to happen immediately. Why is this so crucial? Because it prevents "blocking the line". We all know the pain of a broken commit reaching the main branch. Suddenly, the entire branch is blocked for everyone else! No one can safely deploy, no one can merge, until someone clears the "blocker". What should have been a localized little mistake turns into a massive team-wide stoppage. Quite the waste of time, isn't it?

Because of this automated gatekeeper, by the time a tester actually receives a build nowadays, they can trust that the foundation is solid. This frees them up to do what they do best: deep, exploratory work, rather than wasting time triaging trivial blockers that shouldn't have made it to the test environment in the first place.

But here is the catch: this entire setup only works if these tests are 100% reliable. A "flaky" smoke test is actually worse than having no smoke test at all! Why? Because it trains your development team to just ignore pipeline failures. If a test in this suite fails even once for a reason other than a genuine bug, you need to demote it or delete it immediately. The goal here is high signal and high speed. If your smoke test is unreliable and slow, it shouldn't be there! Let's be real: if you can't trust the smoke, you can't trust the gate.

PRODUCTION SMOKE TESTING VIA SYNTHETIC MONITORING

Imagine your core payment service silently failing at 2:00 AM on a Sunday. With traditional, passive monitoring, your system just sits there waiting for a real user to encounter the error and trigger an alert. If the next customer doesn't try to buy something until 9:00 AM, you've just racked up seven hours of

assumed downtime without anyone even knowing it. Quite a massive risk for your SLAs, isn't it?

Because of that, we use synthetic monitoring today. It's essentially a production smoke test that runs continuously around the clock. Instead of waiting for real users to stumble over a broken environment, it actively injects simulated user journeys—like logging in or checking out—every couple of minutes. It acts as a lightweight alarm system that tells you immediately when the main flows are broken.

Now, it is very tempting to just take your existing automated UI tests and point them at the production environment. When teams try this, it rarely ends well. UI tests are heavy, slow, and notoriously prone to flakiness. They might break because of obscure browser quirks, a weird popup, or minor rendering issues. You really do not want to wake up an on-call backend engineer at 3:00 AM just because a CSS library failed to load on a simulated mobile device. That just leads to alert fatigue, and soon everyone ignores the smoke tests entirely.

Instead, a much more robust and pragmatic approach is using service-level API probes. You basically capture the core backend requests of a user journey and run them as a lean, sequential collection. While a full UI test might take minutes to complete, an API probe usually finishes in a matter of seconds. Because they are so incredibly fast, you can run them constantly. You can even configure them to only trigger an alert if they fail two or three times in a row. If that happens, you know for sure it's not a flaky test—you have a genuine production fire, and you caught it before your users did.

Trust the Machine, Start Exploring

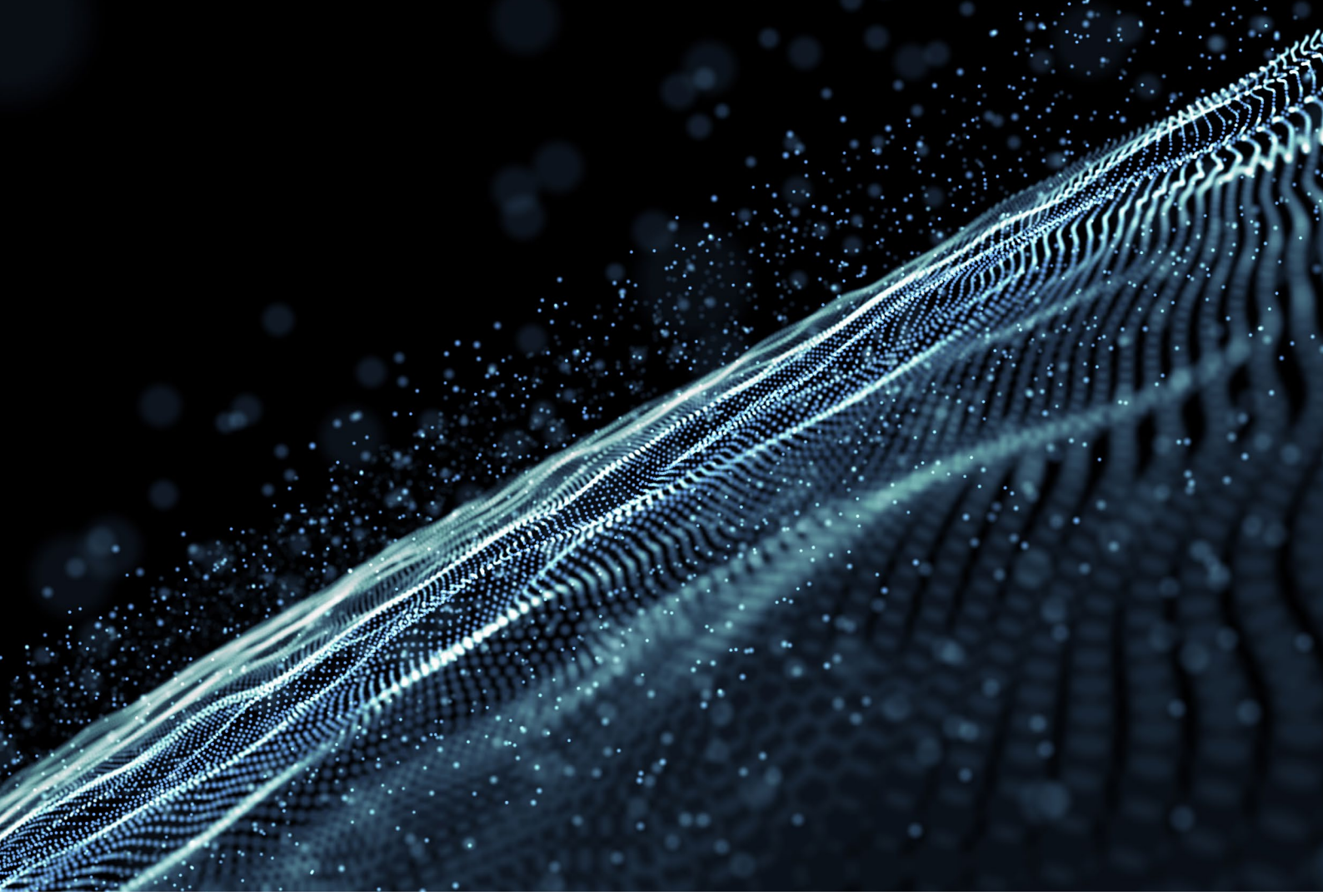
Smoke tests aren't gone; they have just evolved from a manual, time-consuming chore into automated, continuous guardians residing in our pipelines, deployments, and production monitors. The days of a dozen testers sitting idle, waiting for a green light just to begin their actual work, are—and should be—long behind us.

If you still find your team doing a "quick manual smoke test" before every formal test phase, you are likely compensating for gaps in your CI/CD automation or dealing with a fundamental lack of trust in your test suite. Automation shouldn't just mimic old manual processes; it should eliminate the need for them entirely.

Machines are incredibly efficient at answering the basic question, "Is it completely broken?" Humans, on the other hand, excel at finding out, "In which ways is it broken?". By folding that initial confidence check into small, sharp, automated gates, you stop wasting human potential on repetitive sanity checks. Instead, you free up your testing professionals to focus on the deep, exploratory, and edge-case testing that truly cannot be scripted.

Take a hard look at your team's release process: Are there manual bottlenecks masquerading as necessary "smoke tests"? Identify just one core business flow you still check by hand after a deployment, and make it your goal to script it into your pipeline or canary checks. Stop testing whether the application simply turns on, and start giving your team the time to test how well it actually works.

Christian Baumann
Principal Test Architect
@ MaibornWolff GmbH



Textile Tech to AI Anxiety

What testers can learn from textile history

Author: Ester Greenwood

TEXTILE AUTOMATION

Weaving is almost 30,000 years old, and therefore possibly the oldest binary system. In its nature, weaving is binary because whether a thread goes under or over decides what thread shows on the good side (the front-end, so to speak).

With different colors, a weave could be used as a canvas. Patterned fabrics were more valuable than plain weaves. So the Chinese brocade weaving became popular worldwide.

A traditional brocade loom requires two workers: a weaver and a drawer. The drawer would sit on top of the loom and pull up threads according to a pattern. In Europe, this was often a young boy, pulling strings for 16 hours a day.

In 1804, Joseph Marie Jacquard developed his Jacquard loom. This loom uses a stream of punch cards, sewn together. Every single warp thread was now separately programmable. Jacquard's loom was not the first automated brocade loom, but it was more reliable and cheaper to program than others.

A CASCADE OF INVENTIONS

The Jacquard loom took Europe by storm. It was an absolute spark of inspiration to Charles Babbage. Known as the 'father of the computer', his inventions used punch cards as input. Then, base-10 gears would make mathematical operations that were printed on paper. His 'Difference Engine' and 'Analytical Engine' were not built in full until the late 20th century.

Babbage worked closely with mathematician and writer Ada Lovelace. Lovelace was high-born, educated, and equally matched to Babbage's genius. She published the first computer algorithm- instructions on how the Analytical Engine could calculate Bernoulli numbers. She also speculated that in the future, computers would be used to make art and music. Her ideas are regarded to be the groundwork of modern programming.

BINARY PROCESSING

Jacquard's invention was the start of a binary revolution. The punch card proved useful in the US census of 1890. Herman Hollerith had designed machines to process and sort the cards. The system saved 5 million dollars and years of labor. Hollerith and his inventions were later bought by the company that became IBM.

While the punch card allowed for binary storage, George Boole wanted to formalize binary reasoning. His Boolean algebra (1847) was initially theoretical. Nearly a hundred years later, Claude Shannon proved that electrical relay switches could represent Boolean Logic. The foundation for electronic processors was laid.

Punch cards of a Jacquard loom at the London Science Museum

"The Analytical Engine weaves algebraic patterns, just as the Jacquard loom weaves flowers and leaves."

– Ada Lovelace

HAND-MADE MEMORY

The common thread of textile and tech goes further, even as far as the moon. With the Apollo missions, NASA was in need of a stable, lightweight ROM (Read-Only Memory). Core Rope Memory was chosen, as it could withstand the shocks, radiation and temperature changes of space travel.

NASA employed expert textile workers to 'knit' these ropes. It gained the nickname of LOL Memory, for 'Little Old Lady'. It had to be bug free because it could not be updated. MIT designed a machine to automatically test the memory, bit by bit. These ladies held in their hands the lives of the astronauts as well as the outcome of the space race.

> *"intricate, challenging handiwork - comparable to weaving - was just as essential to putting men on the moon."*
>
> *– Joy Lisi Rankin*

A worker programming Core Rope Memory (Transistor Museum.)

THE LUDDITES

The amazing inventions of the early 19th century also had their repercussions. Steam-powered looms are fast, but also heavy and relentless. Children were actively recruited. Their small bodies and fingers were useful to fix issues without having to stop the machines.

In Nottingham around 1811, workers rallied in the name of a fictional weaver named 'Ned Ludd'. Lacking formal representation, they smashed machines to get attention. The revolts spread across England and lasted 5 years. Many Luddites were caught, and punished by execution or penal transportation.

The fear of job loss due to technological advancement was named the 'Luddite fallacy'. However, this overly simplifies the point of the Luddite revolts. Many of them were already masters of the new machines. What they fought against was not automation, but exploitation.

TEXTILE TODAY

Unfortunately, the textile industry is still problematic. It requires a huge amount of resources and produces a lot of waste. Labor practices in the developed world have vastly improved since the Luddites. Elsewhere, exploitation and child labor are still common practice.

Today, textile is cheap and widely available but of poor quality. A 'Fast Fashion' garment will fall apart after only a few wears. With the rise of AI-generated code, some people worry that software quality will also decline. Quick to make but essentially of poor quality, dangerous even.

PREDICTING THE FUTURE

Concerns about the power of computers are not new. Even in the year 2000, Bill Joy predicted that robots with human-like intelligence would be built. The robots, controlled by the rich, would make humankind an endangered species.

Joy was called a 'Doom and Gloom technofuturist'. But many doomsday articles have actually undone their own prophecy by inciting action. This makes it seem as though they were wrong. But in many cases, collective action was really what saved us.

The truth is that all inventions have had incredibly diverse effects. Most of them not exclusively good or bad. In the same way Babbage was inspired by Jacquard, one can barely imagine what some of our modern-day geniuses will accomplish with AI.

THE ROLE OF THE TESTER

The AI savvy tester may accelerate their game by using tools responsibly. LLM (Large Language Model) tooling can script and visualize, and convey necessary information about the product faster.

Those who are concerned about AI use may help their case by experimenting. In general, saying 'AI is just bad, okay' is not a strong argument. If you are not willing to use tools by certain suppliers, you might investigate and support alternatives.

Quality characteristics such as reliability might also be a concern. Test your hypotheses by experimenting and presenting the data. AI can be a great help in producing test data and visualizing results. Since LLMs are a relatively new tool, investigating is both valuable and fun.

LESSONS LEARNED

From loom to LLM, our modern lives are almost unrecognizably different. Think on how quickly we lost the skills of making, mending and altering clothes. As testers, our curiosity and critical thinking skills are our most important asset. Overly relying on AI tooling may eat away at those essential skills.

Let us keep our brains active and healthy so we can keep asking those difficult questions. Coincidentally, knitting has been shown to protect your brain from dementia and depression. So pick up those needles, and if anyone asks... You're just practicing for your next job at NASA!

"We know the past but cannot control it. We control the future but cannot know it."

- Claude Shannon

Ester Greenwood
Test Engineer
@ Sopra Steria

AGILE TESTING DAYS

Nov. 16 - 19, 2026
Potsdam, GER & Online

8 Keynotes. 9 Voices. One Stage

Quality, AI and leadership — live from the people rewriting what software testing means. Four days, one plenary stage, zero fluff.

Elisabeth Hendrickson

Consultant @self employed

Joel Tosi

CTO & Co-Founder @Dojo & Co

Rob Myers

Principal Instructor & Coach @Agile Institute

Clare Norman

Senior Quality Coach @Co-op

Manu Cohen

Distinguished AI Architect @Advisor360

Tariq King

CEO @Test IO

Janna Loeffler

Director of Quality Engineering @The Walt Disney Company

Keith Klain

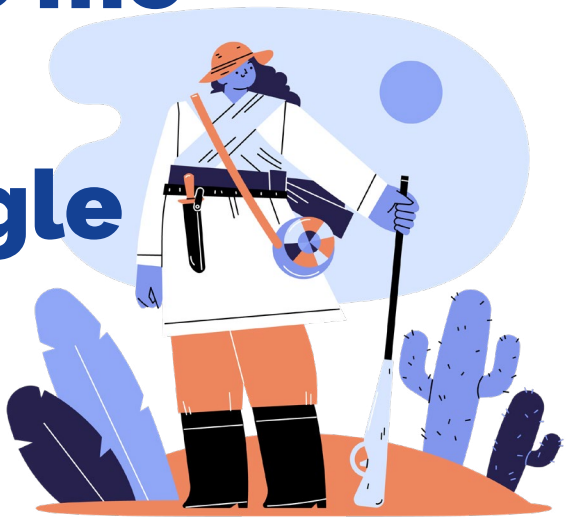
Director, Quality Engineering @KPMG UK

Ray Arell

CEO @nuAgility

SAVE BIG TIME WITH THE EARLY BIRD DISCOUNT. AVAILABLE UNTIL SEPTEMBER 20TH, 2026

Autoethnographic research made me look at testing from a new angle



Author: Jenna Charlton

One of my favorite things about being a tester is the way testers find inspiration and techniques from outside of the testing world and integrate them into their work. In my graduate work I've been discovering the many ways that traditional research techniques overlap with testing behaviors, my favorite of which is autoethnographic research.

What in the world is autoethnographic research you ask? Let's start by looking at ethnography, the research methodology this technique grew out of. Ethnography or ethnographic research is a research method typically used in the social sciences in which the researcher embeds and immerses themselves in a group, community, or culture. In ethnography we acknowledge that researchers will always have a personal slant or bias and instead of attempting to remove bias, instead seeks to learn and empathize with the subjects of the study. Some of the ways we document our findings using this method include journaling, field notes, and narratives of our experiences and observations.

Autoethnography takes ethnographic research one step further and uses the researcher themselves as a subject in the study. Often used in the social sciences, the researcher interrogates their own feelings and experiences in a group, community, or culture. This technique is often used when a researcher is studying their own culture or community, or when they are conducting research on a study of only themselves. Autoethnographic research also uses journaling, field notes, and narratives to document observations during the study.

A couple of real life examples of this technique used in action are:

- In James Osben's *A day in the life of a NHS nurse in 21st Century Britain: An auto-ethnography*: James wrote a narrative of his day as it happened. He then used critical analysis to find the overarching themes of feelings, tasks, and behaviors in the narrative of his shift. Osben's aim in his research was to use "reflective practice." Reflective practice is one of the ways a researcher reflects on, contextualizes, and reframes their experiences for their research.
- Jay Johnson Theil explored her place in academia using ethnographic methods in her research *Working-class women in academic spaces: finding our muchness:* Theil used photography, reflective journaling, movie quotes, and storytelling as research methods, then applied critical analysis to better understand how she really feels about her role in academia.

AUTOETHNOGRAPHIC RESEARCH WORKS A LOT LIKE EXPLORATORY TESTING

If I've explained this well enough you may already see what I saw when first learning about this technique — autoethnographic research sounds like exploratory testing! I've long advocated for paying close attention to your thoughts and feelings as you work your way through an exploratory testing session. With this research method, you can take your exploratory testing even further. Here are a few ways to apply this method to your testing.

Journal your feelings while testing. As you work your way through whatever you're testing, focus on how you feel and what you think. Do you feel frustrated, delighted, or confused? Whatever feelings the application elicit from you, write them down as you feel them. You may also consider other types of journaling using methods like color coded or face sentiment trackers.

Use voice narration. Narrate your thoughts and feelings to a voice app or Zoom session. To make sharing the narrated session easier for your team, you can use a transcription app or speech to text.

Reflect at the end of testing sessions. If you're more interested in overall sentiment as opposed to thoughts and feelings in the moment, consider writing or narrating a reflection of thoughts and feelings from the testing session.

Debrief with your peers. Many teams hold debriefing sessions on a regular basis after exploratory testing. These sessions are the perfect time to share your thoughts and feelings documented in the session.

I will be the first to admit that I can't remember what I did yesterday. Over time, I tend to forget my feelings towards the application I'm testing, which can hinder quality because my brain isn't attune to looking for trends. Starting to use these autoethnographic techniques on an individual level will help you uncover trends in your reactions, thoughts, and feelings related to the application while you test.

You'll have an even more impactful and beneficial experience if you use these techniques with your team and track over time. Some ways to do so are:

Bug summary journal: Keeping a log of bugs you've found and experienced in your application along with sentiment tracking towards each bug helps demonstrate the impact on quality that you feel each bug has. This will also help you see if there are periods when you feel bugs are related to sloppy code or a lack of understanding of the problem the team is trying to solve.

Daily journaling: Tracking your thoughts and feelings about the application on a daily basis will give you a timeline of your sentiment toward the application. This ongoing log gives you an opportunity to identify trends and track confidence in quality in your application over time.

Daily reflections: Similar to reflections at the end of a testing session, a daily reflection gives you a summary of the day. Over time, these reflections will expose trends and when compared with peers, may expose shared thoughts and feelings about the application that haven't been identified previously.

Daily sentiment tracking: A lighter lift than some of the other options that can still expose trends. Each team member tracks their sentiment towards the application each day using either faces, color codes, or other tracking methods. This simplified tracking can help demonstrate changes in confidence in quality over time.

Integrating these techniques is a great way to help your team shift from testing to quality advocacy and demonstrate your ability to identify the quality indicators that have the biggest impact on the overall health and usability of your application.

Jenna Charlton
Developer Advocate
@ BrowserStack

One Change That Made Our QA Pain Hurt Less

Author: Karime Salomon

The pain we accepted for years

Since the first day of my career in technology, one problem has always been there: documentation.

I still remember that back in 2012, one of the main skills expected from a QA was to write clear test cases and clear information. At least in my first job, working on a high-traffic product in an international company, documentation was not optional. For every release or important feature, we had to prepare it.

We took requirements from the Product Owner, notes from meetings, conversations with developers... and we turned all of that into a "nice document" to support testing. We also had to create detailed test plans, listing everything we believed needed to be covered.

Looking back now... I'm honestly surprised by how much time we spent doing all of that before even starting testing.

And then after regression? Making sure everything was updated again.

The idea was good: prepare information that could be reused in the future. The reality? Most of those documents were never opened again.

Over time, tools evolved. We moved from shared documents attached to Jira to more "structured" solutions like Confluence. Suddenly everything was more organized: templates, links, traceability, test cases connected to tickets. It sounded good and it looked better but... did it actually change how we use our source of truth? Not really.

We still pushed teams to create "good documentation". But in many cases, it wasn't reviewed again. Or someone would say: "This is outdated, let's create a new one." (the best idea to save the moment right?).

And the cycle repeated but at the same time, something else changed, a new trend in technology started to happen... people don't stay 10 or 20 years in the same company anymore. Teams rotate constantly. New engineers, new leads, new managers. That's when historical information is supposed to help the most.

We used to say things like:

- "Check the Jira ticket."
- "Everything is documented."
- "You can find it in Confluence."

But onboarding still depends heavily on people. We still schedule calls. We still interrupt others to explain context. So the real question was:

- Is our documentation actually helping? Or are we just maintaining it because we are supposed to?

And then there's another reality: we have too many tools... all good all authorized for the same purpose... storing information.

Even if we say, "Let's keep everything in one place," finding the right information is not easy. Who has time to search properly, validate if something is still correct, update it, and keep it clean? I am not sure how you or your teams work on this point but in my case we are always in a rush. Everything is urgent.

So we keep creating more content just to avoid losing time in the moment... until of course something goes wrong. Yeah you can imagine what?

- An audit.
- A production issue that costs a lot (money... customers... reputation).
- A critical mistake.

That's when all of our information suddenly becomes important again. From my experience, this combination of situations creates what I call a real QA pain. Because even if we are not the only ones responsible for this source of truth, we are usually the ones expected to make it visible, usable, and reliable... and that's definitely not easy.

The AI hype felt familiar

When the AI hype started, from day one, many people were excited about AI. Some people were reading news, reacting to posts, and waiting for the magic pill that would transform everything. Other people around me were trying tool after tool, model after model, always looking for the next thing.

Meanwhile, expectations from companies and executives kept increasing. But the day-to-day people in technology were struggling to understand which tool they should learn, whether AI would take their jobs, and what would be expected from them next.

I saw a lot of emerging opportunities, but also a lot of confusion. AI hype, business expectations, executive pressure, and real team needs were all mixed together.

And I have to admit: I was also part of that selected group!

I was taking courses after work, learning about new tools, trying to understand how we could use AI, where we could use it, and what value it could bring. But sometimes, by the next week, what I had just learned felt outdated because there was already a new tool doing something better.

So at some point, I decided to pause. I needed to stop searching for trends and understand what really mattered.

And from my QA perspective, this felt like déjà vu.

Back in 2013, when I had the opportunity to start working with automation, that word was also a hype, of course not as AI hype which impacted a variety of roles but it was really really similar. Every manager and executive wanted everything automated. Automation was seen as the key to success for any team's releases and deliverables timeline.

For sure, automation became very important for CI/CD, faster releases, and better early feedback. But in the beginning, many teams didn't really know what they wanted. Changing automation tools was too common. Companies were stuck in a loop of trying to find the "perfect tool".

So when AI started becoming the new big thing, I thought:

> Wait a second. I had seen this before.
> The same rush.
> The same pressure.
> The same search for the right tool.

But automation only really started to work for my teams when we stopped focusing on the tool, or on the number of automated test cases, and started focusing on the value automation was bringing to the team, the product, and the quality of the product.

So I asked myself the same question again:

- What is the value I want to get from AI?

Stop looking for tools, start looking for solutions

If the key to automation success was not the percentage of test cases automated, but the real value it brought to the table, the critical bug found quicker, the broken link identified in minutes and the whole critical path verified automatically without having people extending their testing time in a Friday night just because we release on Monday... then maybe AI needed a similar approach.

That's when I stopped and started a retrospective in my own journey, instead of trying to find the best tool, I decided to make a list of the problems we actually had. Because when there is a real problem, the solution has a better chance of bringing real value and yes, one of the main problems I identified was as you can imagine... documentation.

I was honestly frustrated with how we were handling it. And of course, AI looked like something that could finally help fill that gap.

My first experiment failed

At the beginning, I worked on a project building my own RAG using LangChain, ChromaDB, Python, and a local model as a proof of concept. I faced a lot of challenges.

The data had to be in a specific format. The chunks were not always saved correctly. I needed to track versions of files. Otherwise, I was still in the same place: dealing with outdated information. On top of that, I had the extra work of defining clear formats for every single intent I needed, it was frustrating, it was not giving me the correct "conversation with your documentation" that I was expecting with this kind of tool.

Then I found NotebookLM. I'm not saying this is the only solution, but it definitely worked for me.

In the initial versions of NotebookLM, I discovered that we could finally do what I was expecting: talk to our knowledge base even if we are not following a strict format for it. We could ask complex or basic questions, as many times as we needed.

The tool had the versatility I was looking for, so I decided to build an initial proof of concept. One of my teams was constantly struggling with the learning curve and the onboarding process. Sometimes we think all tech projects are greenfield projects or startups with almost no documentation. But in my journey, I have faced the opposite many times: never-ending, old, outdated shared understanding spread across multiple places.

After some experiments focused on onboarding, the magic started to happen... but of course, nothing works without effort.

You know the quote: garbage in, garbage out. And I proved it. I really did.

In the initial stage of my proof of concept, I actually considered it a good starting point because of the versatility of the tool. I was adding PDFs, text documents, Google Docs, even slides from onboarding demos. Everything looked promising.

The problem? A lot of that information was outdated.

The architecture had changed so much over time that even if I could "talk" to the documentation, many answers didn't make sense. Sometimes the information was technically correct, but no longer useful in the current context. So we needed to spend time reviewing documents, identifying what was no longer applicable, and cleaning things up.

And honestly... cleaning the house is not always easy.

At the beginning, we also assumed people would naturally use the tool because it was helpful. They didn't.

Some team members still preferred asking another person instead of searching. Sometimes updating information was skipped because it was "faster" to continue working. That was another lesson learned: access to information is not enough. Habits matter.

Little by little, as we cleaned information and defined the right context, we started seeing better results. It took time... I won't lie. We even went through our own testing phase while cleaning and adjusting the knowledge base... yes we are always testing :)

But eventually, it started working. And for the first time, we could actually see the value: not in the tool

itself, but in having organized, usable, and living knowledge. Finally... a good result.

When I took the time to organize just a piece of the problem, I went back to my original thought: first identify the problem. The problem was documentation. But then I broke it down into smaller parts:

- onboarding
- QA process, tools, and generic information
- troubleshooting
- monitoring
- testing support
- requirements

When I started joining pieces of different problems inside the big one, suddenly we were able to really talk to our source of truth. And not only QA could use it. Tech stakeholders, production support, and new people in the team could also ask questions and get useful answers.

Then we changed one important thing:
We made it part of the process.

Now, with AI tools, there are many ways we can use them for our benefit. But we always need to look at the same north: **the real value**.

The One change that actually helped

The biggest improvement didn't come from NotebookLM itself. It came from one simple decision: we stopped treating documentation as something we create when needed and started treating it as a living part of our process.

Instead of relying on people to remember information, we made knowledge maintenance part of the workflow. Every feature, investigation, onboarding activity, troubleshooting guide, monitoring query, or important decision had a place in our shared source of truth.

Most importantly, updating that knowledge was no longer optional. We built it into the process.

Our flow moved from something like this:
Jira in QA → Search old documents → Ask teammates → Search Confluence → Test → Close ticket

To this:
Jira in QA → Gather information → Load/update living documentation → Test → Capture new learnings → Close ticket

For some people, this may sound obvious. But for us, it was a small change that made a big difference.

In the past, nobody really cared about real information until we needed it. And as QA, we paid the price when something went wrong. When you have four different versions of the same document, with different references, you know you need something better.

I would say we implemented interaction with our living documentation as part of our process.
If we have a question, we don't go to five different sources anymore. We don't need a new person asking the same basic questions multiple times in different contexts. Terminology is more consistent across the team. And finally, we were able to standardize how we use that knowledge.
Even for simple things like:

- Where are the servers?
- What is the IP address?
- Which query do we use for monitoring?
- Which link should I use for X, Y, or Z?

Now everything is part of that shared source of truth.

Standardizing this was not easy. We started small: first with one person, then two more, and then the whole team.

Results beyond documentation

Later, we measured that onboarding time was reduced by around 50%. But honestly, the most noticeable change wasn't the number itself.

Before, a new team member would spend days jumping between Jira tickets, Confluence pages, old documents, and multiple meetings with teammates just to understand the basics of the product.

Now, one of the first questions we ask is:

"Did you check the source of truth?"

Instead of scheduling another call, people can start by asking questions, exploring the context, and understanding the terminology on their own.

We also noticed a significant reduction in repeated questions coming from business and support areas. Questions like:

- Which environment should I use?
- Where can I find the monitoring dashboard?
- What is the purpose of this feature?
- Has this behavior changed before?

Many of those answers were already available, accessible, and consistent.

And even though we are still exploring new tools and investigating how to accelerate AI adoption across the SDLC, for this particular problem we found something that worked.

The biggest difference for me is simple: I no longer feel the same frustration when I need information about the product.

I know where to look.
I know where the information lives.

And most importantly, I trust that what I find is accurate and relevant.

Are you also rushing to find the right tool?

If so, maybe it is worth stopping for a moment and analyzing your real pain point first. We don't always need to start by talking about a tool. We need to talk about the process.

- Which process do we need to improve?
- Which problem do we want to solve?
- Which tool can actually support that?
- And how do we show value so the team follows the change?

Because a tool is always replaceable. But a process can persist if people see the value, if the right changes are applied, and if the team develops the discipline and mindset to make it work.

The Lesson I Didn't Expect

I still believe documentation will continue being one of the biggest pain points in technology. Now we see AI writing test cases, reviewing them, executing tasks, generating summaries... but to make any of that useful, we still need context.

And context is still, many times... documentation.

Organizing knowledge, keeping it updated, standardizing information across teams, that still requires discipline. AI can accelerate how we consume and structure information, but it doesn't remove the effort of maintaining it.

Maybe that's the biggest lesson I learned through all of this. The value was never in finding the right tool... The value was in finally changing the habit.

Karime Salomon
Senior Director, Quality Engineering
@ IDT Corporation

Practical Principles for Agentic QA

Navigating the New Frontier of Quality Assurance in the Age of AI Agents

Author: Manu Cohen-Yashar

You are a software Quality Assurance professional. You've been testing complex software for decades. You've weathered the storms of countless new technologies and trends, adapting, evolving, and consistently ensuring applications meet the quality standards that businesses expect and rely on.

Then comes AI...

New paradigms. New challenges. New frontiers.

And with them, a question that keeps you up at night: What do I need to do to maintain the same level of quality and trust I've delivered for years? What's still valid and what has been deprecated? Or put simply: what's next?

Here's the good news: Yes, agentic AI is new and exciting. Yes, it introduces unique quality assurance challenges that we'll explore in this guide. But here's what matters: **agentic applications are**

still software, and software must be validated. All the knowledge, experience, and practices you've accumulated over your career? They're now more important than ever.

Some practices will need adaptation. Some will require a new focus. But at a high level, everything you've mastered remains valid and absolutely essential. What we're adding is a new layer of practices and principles specifically designed to evaluate the intelligence layer that makes agentic applications unique.

Is My Job Safe in the Era of Agentic AI?

Absolutely.

You're a great QA professional, and you know better than anyone the critical role that creativity plays in quality assurance. You think outside the box. You test scenarios that developers never considered. You find the edge cases that would slip past automated checks. You ask the question: "What could go wrong?" in ways that no checklist can capture.

Creativity is a fundamentally human quality. Sure, AI can generate outputs that appear creative by recombining existing patterns, and sometimes these outputs are genuinely useful. But true ingenuity? The kind that creates something completely new, that breaks conventions, that thinks in ways no one has thought before? That level of thinking is distinctly human.

This is exactly the type of thinking that great QA engineers bring to the table. Research in cognitive science confirms that human creativity involves not just pattern recognition, but the ability to make novel connections across disparate domains, to understand context in nuanced ways, and to imagine failure modes that have never occurred before. QA is more than engineering; it's original, it's art. Your job isn't just safe; it's irreplaceable.

The Foundation Still Stands

Let's start with what hasn't changed. Agentic applications are software applications. Period. They have deterministic components and many nondeterministic building blocks, but at their core, they're still software.

Think about it: Much of the infrastructure and code flow that builds agentic applications relies on the same frameworks you've been working with for years (Python, Go, C# libraries) and familiar infrastructure (file systems, communication stacks, databases, hosting clusters). These aspects haven't suddenly become exotic just because there's an LLM in the mix. They must be tested, and all the tools and practices you already own still apply.

Agentic applications have users. They have basic performance requirements, assumptions, and goals. These must be tested. Proven practices such as User Acceptance Tests, Sanity Tests, Performance Stress Tests, and System Tests, *all these great, battle-tested practices still apply*. Maybe your test plan needs some modifications to account for the probabilistic nature of AI, but the fundamentals remain: we still need to ensure that basic user expectations are met and that all system components work in concert, regardless of which implementation engine powers them.

The Intelligence Layer: What's Actually New

Here's where things get interesting. AI introduces what we call the *intelligence layer*, and it brings new evaluation goals to the table.

On top of all the traditional evaluation goals, we now need to measure that the application doesn't drift and continues to operate in accordance with our intelligence guidelines while delivering trustworthy results. The challenge? Intelligence isn't well-defined, trust is subjective, and the building blocks of agentic execution are stochastic by design.

In the world of agents and probabilistic models, the classic "Given-When-Then" assertions are insufficient. We need to think differently about evaluation. Not *completely* differently, mind you, just differently enough to account for the probabilistic nature of the intelligence layer.

Principle 1: Understand the Cognitive Cycle

Understanding the flow of agentic execution is essential for building an effective test plan. At its

core, the cognitive cycle consists of four stages: perception → reasoning & planning → execution → reflection. But here's the twist: it's rarely sequential. Instead, it often loops and branches, creating what we call the **cognition loop**. For example: perception → reasoning & planning → execution → reflection → perception → reasoning → planning → execution → reflection, and so on.

PERCEPTION

This is where the agent is triggered by an initial prompt, question, or piece of information. During perception, the application constructs context (also known as context engineering) based on company knowledge (often using RAG pipelines), arguments, memory, available tools, and any other relevant information. The output? A complete prompt ready to be sent to the LLM.

REASONING & PLANNING

This is when the LLM "understands" the problem and evaluates the information presented to solve it. The result can be an explicit plan presented to the application for approval, or an implicit plan that the agent keeps in memory and simply follows. Typically, the agent determines which tools from the perception phase are useful and incorporates them into the plan.

EXECUTION

The agent executes the plan. It generates tool calls that activate tools and provide more context to the model. It performs cognitive processes that LLMs excel at, such as summarization, analysis, or any other form of text generation.

REFLECTION

The agent "checks" whether it succeeded. Sometimes, it recognizes failure and attempts to correct itself by triggering another cycle of perception, reasoning, and execution, creating a feedback loop. Reflection is when the agent self-corrects and recovers from unintended consequences.

BUILDING YOUR TEST PLAN

Now that we understand the cognitive loop, we can build a comprehensive test plan that ensures agents' reasoning is correct, evidence is grounded, actions remain in scope, and decisions comply with policy and safety standards. Your test plan should cover these dimensions:

- **Reasoning correctness** — Does the agent demonstrate logical and consistent thought processes?
- **Evidence grounding** — Are outputs traceable to verifiable sources?
- **Execution fidelity** — Do actions match declared plans?
- **Resilience** — Does the agent recover gracefully from failures?
- **Safety and compliance** — Does the agent adhere to rules, ethics, and regulations?

Principle 2: Adopt the Stochastic Mindset

Agentic QA calls for a stochastic mindset, and I'll be honest: this can be challenging. As QA engineers, we're trained in deterministic thinking. We love our precise assertions, our exact expected values, our binary pass/fail results. But since the building blocks of agentic applications are probabilistic models, our evaluation methods must adapt.

Agentic behaviors can be measured using well-defined metrics, but as QA engineers, we need to consider their distributions and statistical properties. We're moving from deterministic assertions about specific values to conditioning on distributions of probabilistic measurements. It's a shift from "the output must be exactly X" to "the output should fall within this statistical range Y% of the time."

Identifying and developing the right metrics isn't easy, and measuring them is even more complex. But you're a creative QA engineer, so this is where you shine. It starts by understanding your system's expected behavior through a stochastic lens. Every application is different, every use case has different statistical attributes, and every use case has different non-negotiables.

You can start your investigation based on the extensive body of research on AI evaluation, as well as common industry metrics and evaluations built on it. Yet only you, who understands your specific landscape better than anyone else, can define

what's normal and what's not. Here are some examples of metrics you might consider (this is far from exhaustive):

Success Metrics: Task Success Rate, Goal Completion Rate, Partial Success Rate

Accuracy Metrics: Action Accuracy, Tool Use Accuracy, Tool Selection Precision, Factual Accuracy, Groundedness

Efficiency Metrics: Trajectory Efficiency, Step Count, Task Time, Latency per Action, Cost per Task, Token Efficiency

Reliability Metrics: Invalid Action Rate, Hallucination Rate, Recovery Rate, Error Handling, Retry Success Rate

Context Metrics: State Tracking Accuracy, Context Retention, Multi-turn Coherence, Response Relevance

Quality Metrics: Reasoning Quality, Planning Effectiveness, Subgoal Achievement, Action Justification Quality

Safety Metrics: Safety Violations, Constraint Adherence, Harmful Action Rate

User Experience Metrics: Human Intervention Rate, User Satisfaction, Preference Alignment, Helpfulness Score

Optimization Metrics: Exploration vs. Exploitation Balance, Redundant Action Rate, Backtracking Frequency, Dead-end Rate, Progress per Step

Classic ML Metrics: Precision, Recall, F1 Score, Semantic Similarity, Edit Distance from Optimal Path

You got the point — there are many established metrics to work with, but this is only your starting point. Working with your data science team, you can define these metrics mathematically, determine computation methods, and establish thresholds to test against. Open-source projects like DeepEval make this mathematical knowledge much more practical and accessible to engineers, so you don't need to implement the math from scratch. But you absolutely want to understand what each metric is measuring and what it means for your specific application.

Principle 3: Instrumentation Is Non-Negotiable

Statistical analysis requires a sufficiently large dataset. There's no way around it. You need to collaborate with the developers writing the application to ensure they instrument the code to produce the data you need.

Now that you understand which metrics matter, you know what to look for and what to request from developers. The good news? Most developers have already enabled tracing and logging and implemented business metrics for their own reasons. Instrumentation isn't a foreign requirement; it's something they need too. Developers need metrics to prove their work is successful, so it's in their interest to ensure their instrumentation is solid.

That said, developers and QA see the world through different lenses. It's worth having a conversation to make sure all the application behavior data you need is being collected. The encouraging news is that tracing is supported by almost every popular agentic and LLM framework. It's becoming a standard that everyone adheres to. You just need to make sure tracing and logging are configured correctly and implemented in a way that supports your test plan.

For a detailed exploration of this topic, see the companion article "Practical Principles to Make

Agentic Observability Actually Work", which covers observability implementation in depth.

Principle 4: Follow the Cognition Loop

Now that we understand the cognition loop, we can build tests against each stage and element. Below are examples of tests you should develop, organized by cognition stages. As a creative QA professional, you'll undoubtedly invent many more.

PERCEPTION STAGE TESTS

- Is the context properly structured?
- Is the context within acceptable size limits?
- Were user prompts properly sanitized?
- Does context construction generate a dataset with relevant, accurate, and concise information that the model actually needs? (This includes all metrics related to RAG performance.)

REASONING & PLANNING STAGE TESTS

- Is the generated plan appropriate for the task?
- Are the right tools selected with the correct parameters?
- Does the model assign correct meaning to entities and concepts (i.e., mental model accuracy)?
- Does reasoning flow toward a solution make sense, and are they appropriate?
- Do proposed actions adhere to the company's ethical standards and industry regulations?

EXECUTION STAGE TESTS

- Are tools executed correctly and in the right order?
- Does the model produce artifacts in the appropriate domain?
- Do artifacts adhere to the company's ethical standards and industry regulations?
- Does the model properly ground its outputs in the available context data?
- Are results relevant, accurate, and concise?
- Does the model preserve important information from the context without ignoring or altering it?
- Are the results aligned with the agent's goals (as described in the system prompt) as well as the company's and societal goals?

REFLECTION STAGE TESTS

- Does the agentic flow identify its own failures?
- Can the agent accurately describe its failures?
- Can the agent fix its failures through self-correction?
- Does the agent escalate to humans when appropriate?

HOLISTIC SYSTEM TESTS

Together, the cognition loop implements an intelligent system that you want to examine as a whole. You should implement end-to-end tests such as:

- Is token usage appropriate and within budget?
- Is latency reasonable for the user experience?
- Are final results relevant, usable, and safe?
- Is the cost per interaction manageable and sustainable?
- How does the system perform under stress (massive prompts, super complex questions, edge cases)?
- Are user engagement metrics meeting expectations?

Principle 5: Experimentation and Regression Testing

Change management has always been challenging in complex systems. You know this well: you introduce a tiny change here, and everything collapses over there. The butterfly effect is real, and it's frustrating.

Agentic systems? They're complex systems on steroids. The change management challenge is now amplified like never before. We need to consider not only the changes we intentionally introduce, but also changes in production data and their effects on model performance, changes in the models themselves (which happen frequently as providers release updates), and changes in how users interact with intelligent systems.

All these changes beyond our control cause something we call *behavior drift*. Systems that have proven to behave in a certain way after rigorous, long-term testing slowly change their behavior and start to "drift" into uncharted (and often inappropriate) territory.

This is where the metrics you defined to measure your system's statistical behavior become vital. As part of your observability implementation, you should monitor these metrics continuously and alert on any anomalies. Think of it as your early warning system for drift.

THE GOLDEN DATASET: YOUR ISLAND OF STABILITY

Because of the rapid and diverse changes at almost every level of an agentic application, you need to establish a stable point of reference. In a world where everything changes, you need an island of stability. That's the role of the golden dataset.

A golden dataset is a curated mapping between inputs, configuration, and ideal expected outputs. Golden datasets define "normal." They define your expectations and the system's desired behavior. They're created with care by subject matter experts who understand both the domain and the desired outcomes.

You might use production traces as a starting point, but experts should then improve, refine, and augment the dataset to represent ideal execution. Now you have a reference point. This is your beacon. This is your stable island in a sea of constant change.

ARMED FOR CHANGE: THE PRINCIPLE OF EXPERIMENTATION

Armed with golden datasets and well-defined metrics, you can introduce changes with confidence by implementing the principle of experimentation.

Experiments are controlled tests you run on the system using the golden dataset's inputs and the configuration, or the new version of the code that you want to test. You then compare the system's behavior against the expected output of the golden dataset. In a way, it's a classical regression test, but the comparison is against statistical behavior rather than specific discrete values.

A classic example: the constant search for a cheaper, more performant model. Almost every week, new models are introduced, and pricing is updated. Almost every week, there's a new model that could be a good candidate to power one of your agents. Using experiments, you can automatically test your application's performance when using this new model. If all tests pass according to your quality metrics, you can confidently deploy the model to production and potentially reduce costs.

The number of potential changes is huge; there's no way to manage them all manually. Fortunately, most modern observability platforms provide infrastructure to completely automate experimentation*. You just need to develop the golden dataset

*Examples include platforms like Weights & Biases, MLflow, and specialized agentic evaluation platforms such as LangSmith and Arize AI. See platform documentation for experiment automation capabilities.

and the metrics, define the experiment parameters, and the infrastructure handles the rest.

Principle 6: Understand the Deterministic and Non-Deterministic Domains

Let's look at two extremes in the testing spectrum: unit tests and LLM-ass-a-Judge tests.

UNIT TESTS: THE DETERMINISTIC FOUNDATION

Unit tests are designed to be deterministic. When code exhibits random behavior (e.g., datetime generation, GUIDs, randomness), we introduce mock functions to bring it into a deterministic state. This approach is perfect for traditional software.

But when testing intelligence? Mock functions aren't the answer. Unit tests are not designed for LLM verification. They're incredibly useful for verifying that the deterministic code works as expected, but not for the agentic piece.

LLM-AS-A-JUDGE: THE PROBABILISTIC VALIDATOR

Agentic validation often uses an LLM as a judge to implement the evaluator. But here's the paradox: the evaluator itself is non-deterministic. How can we trust it? In theory and in practice, it can hallucinate exactly like the agents it's designed to evaluate.

The judge has to be the foundation of our trust, so how can it be run by a probabilistic LLM? The answer lies in numbers and consensus.

Given the models' stochastic nature, we understand that to obtain an evaluation result we can trust, we must use a *collection of agentic evaluators* and never rely on a single instance evaluation. Agent evaluation frameworks that provide built-in LLM-as-a-judge implementations enable you to create an army of evaluators and use a majority vote (or any other statistical aggregation) to generate the final test result.

In a way, this mimics the strategy we would use with human evaluators. Humans are also non-deterministic in their judgments, which is why we consult groups when we need to trust the result. There's a reason the Supreme Court has multiple justices and why juries consist of a group of people who must decide together on a verdict. The same principle applies to AI evaluation.

Principle 7: Agentic Security and Boundaries

Every company will claim that "Safety is our number one priority." If that's genuinely the case, red teaming must be executed. This isn't optional.

Red teaming requires a group of QA professionals who specialize in introducing adversarial inputs to your system and verifying that it continues to behave safely and responsibly. Agentic AI introduces a variety of new threats we now need to account for, including prompt injection attacks, goal hijacking, and unintended tool usage.

When you plan your QA team, allocate appropriate resources to test agentic security. For a detailed exploration of this topic, see the companion article "How to Secure Agentic Applications", which covers security testing in depth.

COGNITIVE BOUNDARIES

The agent has a goal, and we expect it to fulfill that goal within reasonable boundaries. We expect results only in the relevant domain, tools to be invoked only for legitimate purposes that can be justified, and actions to stay within ethical and regulatory constraints.

Like we set boundaries for a small child ("Don't touch the hot stove," "Look both ways before crossing"), we set boundaries for our agents as well. Red teaming verifies that those boundaries can never be crossed, no matter how creatively an adversary tries to manipulate the system.

The Path Forward

The emergence of agentic AI isn't the end of traditional QA; it's an evolution. Your foundation of knowledge, experience, and battle-tested practices remains invaluable. What we're adding is a new layer of thinking to address the unique challenges posed by the intelligence layer.

You're still testing software. You're still ensuring quality. You're still protecting users and businesses from failures. But now you're also ensuring that intelligence behaves as intended, that probabilistic systems remain within acceptable bounds, and that AI agents operate safely and ethically.

The seven principles outlined here provide a framework:

- Understand the cognitive cycle to know what you're testing
- Adopt a stochastic mindset to measure what matters
- Implement instrumentation to collect the data you need
- Follow the cognition loop to build comprehensive test coverage
- Use experimentation and golden datasets to manage change
- Balance deterministic and non-deterministic testing approaches
- Prioritize security and boundary testing through red teaming

This is challenging work. It requires you to stretch into new domains, learn new concepts, and think in probabilistic terms when your training emphasized determinism. But you've adapted before. You've mastered new technologies, new frameworks, and new paradigms. This is just the next evolution in your already impressive career.

Your creativity, your skepticism, your ability to think like an attacker, your understanding of what users actually need, *these are the qualities that make you irreplaceable*. AI may be changing the systems you test, but it hasn't changed the fundamental value you bring.

Welcome to the new frontier of quality assurance. Your skills have never been more needed.

Manu Cohen-Yashar
Distinguished AI Architect
@ Advisor360

AGILE TESTING DAYS

Nov. 16 - 19, 2026
Potsdam, GER & Online

One Day. One Room. No Slides Left Untouched

Six hours, small groups, senior practitioners. Go deep on the one skill you actually came for — before the conference even opens.

ARTIFICIAL INTELLIGENCE

Testing AI Chatbots for Real

Discover practical techniques for evaluating chatbot behaviour, testing conversations, and uncovering risks hidden in AI-generated responses.

RAHUL VERMA
Chief Strategy Officer for GenAI
@trendig

QUALITY COACHING

How to Change a System

Use systems thinking to identify meaningful leverage points, understand hidden dynamics, and drive change without unintended consequences.

ELISABETH HENDRICKSON
Consultant
@self employed

JOEL TOSI
CTO & Co-Founder
@Dojo & Co

TESTABILITY

How to Test Effectively

Learn how expert testers think about risk, coverage, context, and exploration to design better tests and uncover more valuable defects.

HUIB SCHOOTS
Quality Coach
@Sogeti

PAUL HOLLAND
Principal Consultant
@Testing Thoughts

SUSTAINABILITY

Building Sustainable Quality in the Age of AI

Explore how to balance speed, automation, and AI adoption while building systems and teams that remain sustainable over time.

LAVEENA RAMCHANDANI
Test Manager
@easyJet

TEST AUTOMATION

Build Your Own "Self-Healing" Testing Agent

Build a self-healing testing agent and learn how automation, heuristics, and LLMs can detect and repair broken tests.

GIL ZILBERFELD
CTO
@TestinGil

ARTIFICIAL INTELLIGENCE

Leading Quality in the Age of AI

Develop the leadership skills needed to guide quality strategy, communicate risks, and influence AI-driven organisational change.

FIONA CHARLES
Software Testing
@Quality Intelligence

ANNE-MARIE CHARRET
Consultant
@Testing Times

ARTIFICIAL INTELLIGENCE

Design For Trust: Engineer Agentic Trustworthy Systems

Learn practical strategies for designing secure, observable, and trustworthy AI agents that operate safely and reliably.

MANU COHEN-YASHAR
Distinguished AI Architect
@Advisor360

MASTERCLASS DAY – BEFORE THE MAIN STAGE

DON'T MISS OUT!

Early bird discount until September 20th, 2026

The Bugs Before the Bugs

Why Agile Teams Should Use AI as a Sparring Partner for Requirements Conversations

Author: Tao Chun Liu

Bugs Do Not Always Start in Code

Most software teams treat bugs as something that appears after code is written. A feature is developed, a tester validates it, a defect is reported, and the team begins the familiar cycle of clarification, fixing, retesting, and release pressure.

But in many Agile projects, the most expensive bugs do not begin in code. They begin much earlier, inside unclear user stories, missing acceptance criteria, vague business rules, and assumptions that were never made visible. By the time the tester finds the defect, the real problem may already have existed for days or even weeks.

The code may be working exactly as someone understood it, but not as the customer expected it. In that situation, testing does not discover a coding mistake. Testing reveals a misunderstanding already built into the product. This is what I call the bugs before the bugs.

This Is Not a New Problem

The idea that requirements misunderstandings create defects is not new to the testing community. Agile teams have been working on this problem for years through practices such as Specification by Example, Three Amigos, Example Mapping, Behaviour-Driven Development and Acceptance Test-Driven Development.

These practices all point in the same direction: quality improves when teams have better conversations before building. Examples, shared language, acceptance criteria, and concrete scenarios help teams uncover assumptions earlier.

AI should therefore not be presented as a replacement for these practices or as a magical discovery. A more useful framing is this: AI can become a scalable sparring partner for the conversations we already know we should be having. It can help product owners, testers and developers explore more angles, challenge vague language and prepare better examples before or during refinement.

The Hidden Cost of Misunderstanding

Agile teams move quickly. Product owners explain requirements, developers ask questions, testers prepare test scenarios, and everyone tries to keep the sprint moving. Because Agile encourages collaboration, teams often assume that frequent conversation automatically creates shared understanding.

In reality, people can leave the same conversation with different interpretations. A stakeholder may describe a business rule in familiar language, a developer may translate it into system logic, and a tester may consider exceptions, auditability, and risk. Everyone may agree that the story is clear, yet each person carries a slightly different mental model.

This is where many quality problems begin. The defect is not always caused by poor coding. Sometimes it is caused by unclear language, missing examples, undocumented constraints, or assumptions that nobody thought to challenge.

A Less Obvious Example: Coupon Eligibility Across Channels

Consider a more realistic user story from a retail-style product environment:

As a loyalty member, I want to redeem a campaign coupon across online and kiosk channels so that I can use the same promotion wherever I shop.

A competent team will probably ask the obvious questions: What is the coupon value? When does it expire? Can it be used more than once? Which products are excluded? These are important questions, but they may not be enough.

When AI is given richer context, such as previous incidents, business rules, channel differences, and a glossary, it may surface less-obvious risks. For example, what happens if a user starts a redemption online but completes it at a kiosk? Which system becomes the source of truth if two channels process the same coupon within seconds? Does "same promotion" mean the same coupon ID, same campaign code, or same customer entitlement? If the campaign resets daily, which time zone defines the day? If the kiosk is temporarily offline, should it allow redemption and sync later, or reject the coupon?

These questions are not just a checklist. They reveal potential defects at the intersection of business language, system integration, data consistency, and customer experience. This is where AI can be useful: not by asking the first five obvious questions, but by helping the team explore the messy edges that are easy to miss when everyone is rushing toward sprint planning.

AI as a Requirement Risk Detector

AI can create value for testing teams when used as a requirements risk detector. When given a user story, acceptance criteria, meeting summary, domain glossary, or past defect pattern, AI can help identify areas that deserve further discussion. It can highlight ambiguous words, suggest missing examples, generate edge cases, and propose questions that testers may want to ask during refinement.

The value is not that AI asks questions no human could ask. Experienced testers already know how to challenge ambiguity. The value is that AI can help teams produce a broader first draft of risk questions faster, make hidden assumptions visible, and support less-experienced team members in learning how strong testers think.

In this sense, AI becomes part of the preparation for Three Amigos or Example Mapping. It can help teams arrive at the conversation with better raw material: possible rules, examples, counterexamples, edge cases, and areas of uncertainty.

A Practical Five-Step Approach

Teams do not need a complex AI platform to begin. A simple and repeatable practice is enough.

1 First, select one user story before refinement or sprint planning. The story should be realistic, not artificially perfect. The messier the story, the more useful the exercise becomes.

2 Second, provide AI with context, not just the story title. Include the business goal, user roles, known constraints, Definition of Ready, relevant glossary terms, previous examples, and any regulatory or operational concerns. This is the difference between generic prompting and useful context engineering.

3 Third, ask AI to analyze the story from multiple perspectives: tester, product owner, developer, support team, and end user. The goal is not to accept everything AI produces. The goal is to generate a structured set of questions for the team to evaluate.

4 Fourth, compare the AI output with the team's understanding. Remove irrelevant suggestions, challenge questionable assumptions, and identify the few items that reveal a real gap in understanding.

5 Fifth, convert the refined understanding into concrete testing ideas. These may become acceptance criteria, examples, counterexamples, exploratory testing charters, automation candidates, or risk-based testing priorities.

Prompt Patterns That Produce Better Results

A generic prompt such as "find risks in this user story" usually produces generic output. More useful prompts include context, constraints, and a requested format.

One practical pattern is the Context-Risk-Example prompt: "You are supporting a Three Amigos refinement discussion. Given the user story, business goal, domain glossary, and Definition of Ready below, identify any ambiguous terms, missing business rules, examples, and counterexamples. Prioritize the top five questions that could prevent rework."

Another useful pattern is the Boundary and Failure prompt: "Review this story for integration boundaries, data consistency risks, offline or timeout scenarios, permission issues, and exception paths. Separate must-discuss risks from low-priority edge cases."

A third pattern is the Compare with Past Defects prompt: "Based on the previous defect themes below, review this new story and identify similar risk patterns that may reappear. Explain why each risk matters and suggest one example or test idea."

These prompt patterns work because they do not ask AI to replace the tester. They ask AI to prepare better material for human conversation.

What Can Go Wrong

Being specific about the risks of AI makes the practice stronger, not weaker. The first risk is hallucination. AI may invent business rules, cite non-existent constraints, or overstate confidence. Teams should treat AI output as hypotheses, never as requirements.

The second risk is IP and data leakage. Internal user stories, customer data, architecture details, and incident summaries may contain sensitive information. Teams need clear rules about what can be entered into public AI tools, when to use enterprise-approved tools and how to anonymize examples.

The third risk is refinement bloat. AI can generate forty edge cases when five matter. Without prioritization, teams may drown in possibilities and slow down delivery. A useful practice is to ask AI to classify suggestions into must-discuss, consider later, and probably irrelevant.

The fourth risk is skill atrophy. If junior testers accept AI output, they may stop developing their own questioning ability. AI should be used as a coaching tool: testers should compare their own analysis with AI suggestions, discuss differences, and learn why certain questions matter.

From Defect Detection to Risk Detection

The biggest shift is not technical. It is mental. Traditional testing often asks, "Does the software behave correctly?" That question remains essential. But Agile teams also need to ask, "Have we understood the problem correctly?"

When AI is used early in requirement discussion, testing becomes less reactive. Testers are no longer only the people who find defects after implementation. They become facilitators of better thinking before implementation.

This improves collaboration. Product owners get clearer questions. Developers receive better-defined expectations. Testers gain earlier visibility into risk. The team creates quality together instead of pushing quality responsibility to the end of the sprint.

The Future of Agile Testing Is Better Conversation

If we want better software quality, we need to look beyond the moment when code is ready for testing. We need to examine how requirements are understood, how assumptions are shared, and how decisions are made.

Many bugs are not born in the codebase. They are born in the gap between what one person said, what another person heard, and what the team eventually built.

AI gives Agile teams a new opportunity to inspect that gap earlier. Used responsibly, it can strengthen existing practices such as Specification by Example, Three Amigos, Example Mapping and BDD/ATDD. It can help testers ask better questions, identify requirement risks and create stronger alignment before development begins.

Testing is no longer only about finding defects. It is about preventing misunderstandings from becoming defects. Sometimes, the most valuable bugs are the ones we never allow to exist.

Tao Chun Liu
Founder
@ PMMAYORS

trendig

trendig.com/en/training

GenAI in Testing

safe, effective, and built on sound methods

Most GenAI-for-testing training stops at generation – writing prompts, more test cases, quick test data.

This 3-day hands-on course goes to the part that actually matters: how do you know any of it is good? It's a course about putting GenAI to work in testing without handing your judgment to it.

You'll work through the real testing arc – reviewing requirements, generating and improving tests, handling test data, and sharpening bug reports — with a tester's scrutiny applied at every step.

What you'll take away:

- **Structured prompts that direct, constrain, and shape testing output — not just ask for it.**
- **The habit of improving AI-generated test ideas, test cases, and test data, instead of treating the first answer as the final answer.**
- **Practical handling of the risks that come with AI-assisted work: hallucinations, bias, privacy, and security.**
- **GenAI applied under clear tester oversight, judgment, and accountability — the AI does the work, you own the call.**

Who it's for:

Working testers comfortable with core test-design techniques who want to use GenAI seriously, not just try it.

Register now and put GenAI to work in your testing practice.

A Journey of a Strategic QE: Reinventing Yourself

Author: Zolani Ratya

Introduction

Quality Engineering (QE) has evolved far beyond the traditional model of testing at the end of delivery. Modern organisations now operate in cloud-based, API-driven, highly integrated environments where releases happen faster, customer expectations are higher, and operational risk is more visible than ever before.

For many years, testing teams focused mainly on executing test cases and logging defects. Success was often measured by how many tests were executed instead of how much business risk was reduced. I have seen projects report excellent regression statistics while critical integrations, operational readiness, and downstream system behaviour remained poorly understood.

Testing activity does not automatically equal quality.

Quality comes from understanding risk, business impact, operational readiness, customer experience, and system behaviour end-to-end.

This article reflects on my journey from traditional testing into Strategic Quality Engineering, the mistakes many testing organisations have made, and the mindset shifts required to remain relevant in the future of software delivery.

The Wake-Up Call

The industry changed faster than many traditional QA models could adapt.

Agile reduced long testing windows. DevOps introduced continuous delivery. Cloud migration changed infrastructure management completely. AI started generating test cases, automation scripts, and defect analysis faster than manual teams could respond.

At first, many testers feared automation would replace them. The same fear resurfaced with AI.

In reality, automation and AI mainly replace repetitive, low-value tasks. What remains highly valuable is judgement, business understanding, technical reasoning, and leadership.

I learned this lesson during a large enterprise transformation programme involving cloud migration and highly integrated insurance platforms. Early in the programme, testing teams focused heavily on functional validation and regression execution. The dashboards looked healthy, but production-like performance and downstream integration behaviour were not receiving enough attention.

During one performance cycle, the environment processed dramatically lower transaction volumes compared to production expectations. The issue was not caused by missing test cases. It was caused by weak operational visibility, insufficient production-like validation, and limited observability.

That experience fundamentally shifted my thinking.

I realised that modern Quality Engineering is not about validating screens alone. It is about understanding how systems behave under real operational pressure.

The real threat to QA professionals is not AI or automation. The real threat is refusing to evolve.

From Tester to Strategic QE

The biggest reinvention in my career was moving from a testing mindset to an engineering and business mindset.

Traditional testing asks: "Does the system work?" Strategic Quality Engineering asks: "How do we continuously build confidence across the delivery lifecycle?"

A Strategic QE becomes involved earlier in requirements, architecture discussions, test data strategy, integration planning, automation design, and release readiness. Quality is engineered throughout delivery instead of inspected at the end.

One of the most important changes for me was learning the business deeply.

In enterprise environments such as insurance and financial services, small defects can create major operational and reputational impact. A failed billing process, broken integration, incorrect customer communication, or incomplete claims flow can affect revenue, customer trust, and regulatory compliance.

Technology knowledge gives a QE credibility. Business understanding gives a QE influence.

Many organisations still treat automation coverage as the main measure of maturity. I believe this creates false confidence.

Automation should support engineering confidence, not become another maintenance burden.

The Automation Lesson

One of the hardest lessons in my QE journey came through automation.

Earlier in my career, I believed scaling automation alone would solve delivery pressure. We invested heavily in regression automation and framework growth. Initially, the metrics looked impressive: increasing script counts, growing coverage, and larger execution packs.

But over time, maintenance became difficult. Environments were unstable. Test data became unreliable. Some automated tests generated noise instead of confidence.

The problem was not the tools. The problem was strategy.

Strategic automation focuses on critical business journeys, high-risk integrations, API and contract validation, stable regression confidence, CI/CD quality gates, and production risk reduction.

The objective is not to create more scripts. The objective is to improve delivery confidence.

AI can now generate test cases, summarise requirements, identify patterns in defects, and accelerate automation development. However, AI still lacks business context, operational awareness, and strategic judgement.

The future QE will not compete with AI. The future QE will lead with AI responsibly.

The Technical Evolution of QE

Modern enterprise systems are highly distributed.

A single transaction may travel through APIs, middleware, cloud services, authentication providers, CRM systems, data platforms, event queues, and external integrations.

One of the biggest historical weaknesses in testing was validating only what appeared on the screen. Many downstream failures remained invisible until production.

Today's Strategic QE must understand API testing, cloud platforms, CI/CD pipelines, observability, performance diagnostics, security validation, and production telemetry.

One of the biggest shifts in my own career has been learning observability.

Modern systems fail differently. Transactions can appear successful while downstream queues fail silently, APIs degrade under load, or overnight batches corrupt data.

Tools such as DataDog, Grafana, Dynatrace, and CloudWatch changed how I think about quality. Observability gives teams insight into system behaviour beyond the user interface.

This changes the QE role from reactive defect logging into proactive operational intelligence.

Leadership and Reinvention

Technical reinvention alone is not enough. The future of QE also requires leadership reinvention.

Modern QE leadership requires influence across technology transformation, AI adoption, quality governance, operating model design, engineering maturity, executive communication, and talent development.

One of the most important lessons I have learned as a QA leader is that influence matters more than control.

The strongest quality leaders are not blockers. They are trusted advisors who understand how to balance delivery speed with operational safety.

I remember a conversation with a senior stakeholder during a major release discussion where the pressure to proceed was extremely high. Instead of arguing emotionally about testing status, we reframed the discussion around business risk, operational readiness, and downstream impact.

The conversation changed immediately.

Executive stakeholders respond far better to risk-based language than testing terminology.

Conclusion

The journey of becoming a Strategic QE is ultimately a journey of continuous reinvention.

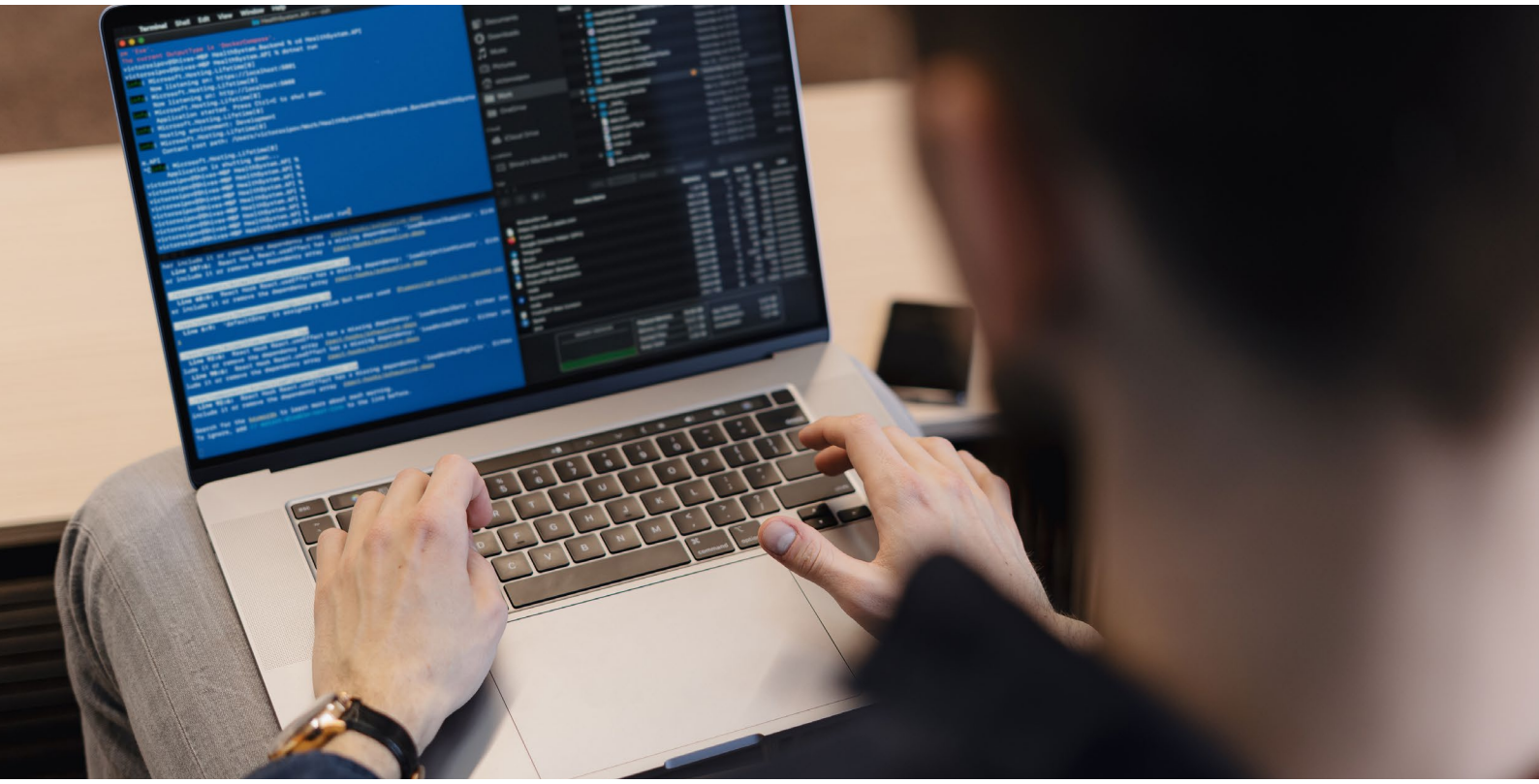
Quality Engineering is no longer only about finding defects. It is about enabling confidence, protecting customer trust, supporting operational resilience, and helping organisations deliver technology safely at speed.

The future Strategic QE is a business enabler, a technology partner, a risk advisor, an operational thinker, and a transformation leader.

The professionals who remain valuable in the future will not be those who know only one tool deeply. They will be the people who can connect quality to business value, combine AI with human judgement, and continuously adapt as technology evolves.

Reinvention is no longer optional in Quality Engineering. It is the job.

Zolani Ratya
QA & Testing
@Santam Limited



The Great Unbundling of QA

Why “the Tester” Is Splitting in Two

Author: José Díaz

For a decade, the anxious question hanging over software testing has been “Will AI replace the tester?” It’s the wrong question. It assumes “the tester” is a single, stable thing that AI will either keep or take away. It isn’t. What we call QA is a loose bundle of very different jobs wearing one title, and AI isn’t deleting that bundle. It’s pulling the glue out of it.

The mechanical parts of testing are becoming nearly free. The judgment parts are becoming more valuable than ever. Those two forces pull in opposite directions, and the role can’t hold together under that strain. The profession isn’t disappearing. It’s de-bundling, and the pieces are re-attaching to very different places.

What’s actually in the bundle

Open up any QA job and you’ll find a dozen unrelated activities stapled together: writing and running regression scripts, exploratory probing, automation engineering, test design, risk analysis, clarifying vague requirements, gatekeeping releases, and shaping quality strategy. The only thing these share is the org chart.

When testing was expensive, bundling made sense. You hired a person to do all of it because the fixed cost of context (understanding the product, the users, the risks) was high, and you wanted to amortize it across every quality task. AI collapses the cost of the cheap, repetitive work to almost nothing.

Once the mechanical layer is free, there's no economic reason to keep it bundled with the expensive, judgment-heavy layer. The glue dissolves, and the activities fly apart.

The market is already pricing this split inside the single job title. Recent industry survey data shows senior testers who move into strategy and quality-leadership work earning a double-digit income premium, while those who stay in pure execution take a double-digit penalty at the *same seniority level*. Same title, opposite trajectories. The market isn't punishing people who test by hand. It's punishing people who *only* test by hand.

This movie has played before, and it ran backwards

Here's the part the hype cycle conveniently forgets: the industry already ran this experiment, a decade ago, without any AI at all.

Around 2014–2015, Microsoft consolidated its development and test organizations under the banner of "combined engineering." In practice, dedicated testers were either converted into developers or let go. In some groups, testing staff had outnumbered developers two to one; afterward the ratio collapsed to roughly one to one, and the test team was quietly renamed "Quality." Google ran its own version, folding the dedicated test-engineer track toward generic software engineering.

The reasoning then is almost word-for-word the reasoning now. Separate testers, the argument went, created a lack of accountability among developers, slowed feedback cycles, removed any incentive to write testable code, and let test architecture drift away from product architecture. Swap "developers" for "AI agents" and you have the 2026 pitch verbatim: *let the people (or the models) who build it also verify it, and dissolve the separate quality tribe*.

But the experiment didn't end where the slide decks promised. It overshot and swung back. Microsoft, having gone from Test, to No Test, eventually started hiring dedicated testing roles again. The retrospective verdict from inside was blunt: the accountability idea had been pushed too far, and the assumption that one person could be both an excellent developer and an excellent tester turned out to be unrealistic. The function didn't vanish. It collapsed, hit the walls of accountability and lost expertise, and partially re-formed at a higher level of abstraction.

That's the pattern worth betting on now: not a clean disappearance, but a violent de-bundling followed by a partial, smarter re-formation. AI will make the swing faster and deeper. It won't repeal the physics.

Half one: quality dissolves downward

The first half of the split pushes quality *down* into the whole team, and, increasingly, into the agents.

"Quality is everyone's responsibility" has been a slogan on agile posters for twenty years and a fiction in practice, because making it true was too expensive. AI finally makes it cheap. The biggest shift of the current moment isn't *how* QA uses AI; it's *who does QA at all*. Developers are now expected to own test automation as a core part of their job, accelerated by coding copilots that generate test code as fast as production code. Unit tests, regression suites, self-healing scripts, synthetic data, first-pass coverage: this entire mechanical layer is being absorbed by developers prompting agents, and increasingly by agents acting on goals rather than scripts.

The forcing function is hard to argue with. The industry has been stuck for years at roughly a quarter of its tests automated, and agentic AI is the mechanism everyone is betting on to finally break through that ceiling. When the analyst firms rename the entire product category from "automation testing" to "autonomous testing," they're naming this exact transfer. The thing absorbing the mechanical work, at the bottom of the stack, often isn't a person at all.

Half two: quality concentrates upward

The second half is the more interesting one, and it's where the new careers are.

As the floor of testing gets automated away, the ceiling rises faster, and concentrates into a small-

er, sharper cadre of specialists who do the work AI can't. The clearest signal is a job title that barely existed three years ago and now appears across job boards: the **AI Evaluation Engineer**. Strip away the branding and the role is defined as turning ambiguous human intent ("make the answers more helpful," "make it safe enough to ship") into measurable quality targets, repeatable evaluation suites, and release gates that catch regressions before users do. It exists for a brutally simple reason: most AI systems fail *silently*, and someone has to build the instruments that detect failure no exception will throw.

That role is the old "test oracle problem," the question of how you decide what *correct* even means when there's no ground-truth answer key, promoted from an academic footnote to a salaried specialty. And it travels in good company. The emerging high-leverage roles cluster around the same theme: eval engineers who design the measurement, prompt and retrieval engineers who own system behavior, agent architects who own multi-agent reliability and trust-and-safety, and AI compliance specialists who own regulatory readiness against frameworks like the EU AI Act and the NIST AI Risk Management Framework.

None of these sit comfortably *inside* traditional QA. They live at the intersection of testing, applied machine learning, product, security, and compliance. But every one of them is, at its core, a *quality* job, and the people best positioned to grow into them are testers who already think in terms of risk, edge cases, and "what could go wrong here."

So the shape of the split is this: the floor rises, because everyone now gets decent baseline testing nearly for free. The ceiling rises faster, because a few people who can specify intent, design evaluations, and assure autonomous systems become disproportionately valuable. And the middle, the pure scripted-regression executor whose entire output was running the same checklist before every release, is what gets squeezed out. Not because that work was unimportant, but because it was the one part of the bundle a machine can genuinely do better.

The forces pushing back

It would be dishonest to present de-bundling as a clean, one-way collapse. Several forces resist it, and they're the same forces that swung Microsoft's pendulum back.

Accountability doesn't distribute. When everyone owns quality, and especially when *agents* own quality, in practice no one is accountable. A model cannot answer to a regulator or a customer for a production failure. A human has to. That single fact keeps a dedicated quality function alive even in the most automated shops.

Independence is a feature, not waste. A developer who tests their own code inherits their own blind spots. So does an AI agent generating tests for the code it just wrote: there's growing evidence that AI-generated oracles tend to certify whatever the implementation already does, quietly baking in the bug instead of catching it. Independent judgment about whether software is actually *right* is precisely the thing fusion destroys.

Domain expertise resists automation. A tester who deeply understands payment fraud, clinical workflows, or aviation safety isn't doing execution; they're doing risk assessment. That's far harder to hand to a generalist model, and it's where the most durable careers will sit.

The headline numbers hide more than they reveal. Government labor statistics project healthy growth for "developers, QA analysts, and testers," but they lump all three into one category, so the figure can't tell you whether the *tester* slice is growing or being hollowed out from within. And manual testing is far from dead: large industry surveys still find the overwhelming majority of testers using it daily. The broad quality function is clearly surviving. The narrow, execution-only job is the one in trouble.

What it means if your title says QA

If your work today is mostly running tests, the honest read is that the floor is coming up to meet you, and you want to be standing somewhere higher when it does. The move isn't to out-type the agents on scripted regression; you'll lose. It's to climb toward the parts of the bundle that are *gaining* value as the rest commoditizes: specifying what "good" means, designing evaluations for systems that have no answer key, modeling risk, probing for the fail-

ures no one wrote a test for, and learning enough about AI systems to govern and assure them.

Think of the future not as a single profession but as a **T**. There's a broad, thin bar of quality literacy that every engineer will be expected to cross: eval-thinking, prompting agents to test, reading coverage critically, reasoning about intent. And there are a few deep vertical spikes (quality strategist, evaluation engineer, AI assurance and compliance specialist, domain risk expert) where the real leverage and the real premium live. The spikes aren't promotions you earn after years of execution. They're different disciplines you can aim at directly.

"Will AI replace the tester?" was always the wrong question, because it assumed the tester was one thing. The tester was never one thing. It was a bundle held together by the high cost of quality work, and that cost just fell through the floor. The bundle is coming apart. The interesting question is which piece you want to be holding when it does.

José Díaz
CEO
@ trendig technology services GmbH


AGILE
TESTING DAYS
THE CONFERENCE
FOR SOFTWARE TESTERS
Nov. 16 – 19, 2026
Potsdam, GER & Online
agiletestingdays.com

REFERENCES

Textile Tech to AI Anxiety
What testers can learn from textile history
By Ester Greenwood

Bach, J. (2025, October). *Seriously Testing LLMs.* [Online]. https://www.satisfice.com/blog/archives/487957

Conni, R. (2011, March). *What the Luddites Really Fought Against*. Smithsonian Magazine [Online]. https://www.smithsonianmag.com/history/what-the-luddites-really-fought-against-264412/

Duguid, P., & Seely Brown, J. (2000, April). *A Response to Bill Joy and the Doom-and-Gloom Technofuturists*. The Industry Standard.

Rankin, J. L. (2022, February). *Core memory weavers and Navajo women made the Apollo missions possible*. Science News.

Schoots, H. (2026, March). *Critical Thinking*. [Online]. https://huibschoots.nl/posts/critical-thinking/

Science and Industry Museum. (2019, June). *Programming patterns: the story of the Jacquard loom*. [Online]. https://www.scienceandindustrymuseum.org.uk/objects-and-stories/jacquard-loom

U.S. Census Bureau. (2024, April). *The Hollerith Machine*. [Online]. https://www.census.gov/about/history/bureau-history/census-innovations/technology/hollerith-machine.html

Practical Principles for Agentic QA: Navigating the New Frontier of Quality Assurance in the Age of AI Agents
By Manu Cohen-Yashar

Cohen-Yashar, M. (n.d.). *How to Secure Agentic Applications*. Medium. [Data Science Collective].

Cohen-Yashar, M. (n.d.). *Practical Principles to Make Agentic Observability Actually Work*. Medium. [Data Science Collective]

DeepEval. (2024). *Open-source LLM evaluation framework. Confident AI*. https://github.com/confident-ai/deepeval

Gabora, L., & Ranjan, A. (2013). *How insight emerges in a distributed, content-addressable memory*. In O. Vartanian, A. Bristol, & J. Kaufman (Eds.), Neuroscience of Creativity (pp. 19–43). MIT Press. https://doi.org/10.7551/mitpress/9780262019583.003.0002

Liang, P., et al. (2022). *Holistic Evaluation of Language Models*. arXiv preprint arXiv:2211.09110. https://arxiv.org/abs/2211.09110

OpenTelemetry. (2024). *Observability framework for cloud-native software.* The Cloud Native Computing Foundation. https://opentelemetry.io/

Perez, F., & Ribeiro, I. (2022). I*gnore Previous Prompt: Attack Techniques For Language Models*. arXiv preprint arXiv:2211.09527. https://arxiv.org/abs/2211.09527

Latest issues

DISCOVER EVERY EDITION SO FAR

trendig

your leading technology services provider

Product Vision
Design Sprints

Consulting, Coaching, Training & Expertise Integration for

Agile Testing Days
Community Meetups

Agile Quality Practices

Agile Transformation & Scaling Agile

Requirements & Backlog Refinement

Software Testing (Automation, Mobile, Performance, Security)

Artificial Intelligence (Applications, Data Services, Testing)

trendig.com