Serge Baumberger

Co-CEO

Zu Inhalten

Serge Baumberger (ehem. Wolf), Co-CEO der Infometis AG (Schweiz), ist spezialisiert auf Quality Engineering und Testautomation in komplexen Enterprise-Umfeldern. Er ist außerdem Dozent (CAS Software Testing/Testautomation), internationaler Speaker und Autor des bei Springer Vieweg erschienenen Buchs „Das Quality Tree Framework“.

Wenn die Pipeline zum Bremsklotz wird: Wie Testautomatisierung skaliert, ohne in der Wartungshölle zu enden

29.04.2026

Dieses Phänomen ist selten ein Problem des gewählten Werkzeugs – ob man auf Open Source wie Playwright setzt oder auf Enterprise-Plattformen wie Tricentis Tosca oder UiPath. Es ist ein Systemproblem. Automatisierung skaliert nicht linear über die Anzahl der Tests, sondern über die Architektur des Gesamtsystems. Wer Automatisierung wie eine lose Sammlung von Skripten betreibt, landet in der Wartungshölle. Wer sie jedoch wie ein erstklassiges Produkt versteht – mit sauberer Schichtung, klarer Ownership und belastbaren Qualitäts-Gates – gewinnt die notwendige Release-Geschwindigkeit.

Ein anonymisiertes Praxisbeispiel aus einem Enterprise-Umfeld zeigt das deutlich: Ein Team mit rund 25 Entwicklern und Testverantwortlichen hatte über Jahre mehr als 1.200 automatisierte Tests aufgebaut, davon rund 70 Prozent auf der UI-Ebene. Die Folge: Pipeline-Laufzeiten von teilweise über drei Stunden, eine False-Positive-Rate von rund 30 Prozent und regelmäßige Diskussionen darüber, ob ein „roter Build“ überhaupt relevant sei. Die Automatisierung war vorhanden – aber sie erzeugte zu wenig Vertrauen.

Das Zielbild: ein Automation System statt einer Test-Sammlung

Ein skalierbares Automation System folgt drei strategischen Prinzipien:

- Fast Feedback First: Ein Test ist nur so viel wert wie die Schnelligkeit seines Feedbacks. Unit- und API-Tests müssen innerhalb von Minuten ein

Signal liefern. Die UI-Ebene ist wertvoll für die User Journey, aber konzeptionell teurer und fragiler.
- Hybride Tool-Strategie: Es gibt kein „One Tool fits all“. Während entwicklernahe Tests oft in Code effizienter sind, spielen Enterprise-Plattformen ihre Stärken dort aus, wo komplexe Business-Prozesse über Systemgrenzen hinweg abgesichert werden müssen.
- Betrieblichkeit (Operability): Automatisierungscode – oder im Falle von Enterprise-Tools die Modell-Logik – benötigt Standards wie Produktionscode: Modularität, Reviews, Refactoring und Versionierung.

Im genannten Praxisfall begann die Verbesserung nicht mit einem Toolwechsel, sondern mit einer strukturellen Frage: Welche Tests liefern früh ein echtes Signal, und welche blockieren die Pipeline nur? Allein diese Perspektive veränderte die Diskussion. Statt „Wie retten wir unsere Suite?“ lautete die Frage plötzlich: „Welche Tests brauchen wir an welcher Stelle überhaupt noch?“

Die kluge Verteilung: so tief wie möglich, so hoch wie nötig

Um die Wartungshölle zu vermeiden, braucht es eine klare Heuristik für die Testplatzierung:

- Unit- und API-Ebene: Hier schlägt das Herz der Business-Logik. Validierungen von Geschäftsregeln und Berechtigungen gehören hierhin.
- Contract-Testing: Ein Sicherheitsnetz für Microservices. Es stellt sicher, dass Provider und Consumer dieselbe Sprache sprechen, ohne dass beide Systeme gleichzeitig hochgefahren werden müssen.
- System- und E2E-Ebene: Hier werden nur die wirklich kritischen End-to-End-Journeys abgesichert.

In der Praxis bedeutet das oft eine harte, aber notwendige Entscheidung: UI-Tests werden nicht weiter vermehrt, sondern reduziert. Im erwähnten Projekt wurden innerhalb von drei Monaten rund 40 Prozent der bestehenden UI-Checks entweder auf API-Ebene verlagert oder ganz entfernt, weil sie kaum zusätzlichen fachlichen Wert lieferten. Gleichzeitig wurden zentrale Geschäftsregeln näher an der Fachlogik abgesichert.

Das Ergebnis nach dem ersten Umbau war deutlich:

- Pipeline-Laufzeit von über drei Stunden auf rund 55 Minuten reduziert,
- Anteil der UI-Tests von ca. 70 Prozent auf unter 30 Prozent gesenkt,
- deutlich weniger Fehlalarme in den täglichen Builds,
- schnellere Rückmeldung an die Entwicklungsteams bereits innerhalb der ersten Pipeline-Stufen.

Die wichtigste Erkenntnis dabei: Nicht mehr Automatisierung schafft Skalierung, sondern die richtige Verteilung.

SIGS.DE

UI-Stabilität durch Architektur: Abstraktion schlägt Skripting

UI-Tests scheitern selten an einem Framework. Sie scheitern an fehlender Architektur. Eine robuste Strategie braucht Entkopplung.

- Code-basiert – Domain Layer: In Frameworks wie Playwright beschreiben Tests Domänenaktionen, zum Beispiel KundeAnlegen() oder KreditFreigeben(). Die technische Umsetzung liegt in einem darunterliegenden Adapter-Layer.
- Modell-basiert – Enterprise Power: Tools wie Tricentis Tosca lösen dies durch Abstraktion. Anstatt Skripte zu schreiben, werden Module erstellt, die die technische Ebene repräsentieren. Die Testfälle selbst bestehen nur noch aus fachlichen Schritten. Ändert sich ein technisches Element, wird nur das Modul einmalig aktualisiert – alle verknüpften Testfälle bleiben intakt.

Ein typisches Vorher-Nachher-Muster in Projekten sieht so aus: Vor der Umstellung passen Teams nach jedem größeren UI-Release Dutzende Tests einzeln an. Nach der Einführung eines Domain Layer oder einer sauberen Modulstruktur wird derselbe Änderungsaufwand zentral abgefangen. Was früher mehrere Personentage band, reduziert sich im besten Fall auf einige gezielte Anpassungen an einer Stelle.

Testdaten „on the fly“: der unsichtbare Flaschenhals

Ohne Strategie für Testdaten ist Skalierung Zufall. Kollisionen in geteilten Umgebungen sind eine der häufigsten Ursachen für Flakiness.

Ein skalierbares Modell unterscheidet:

- Seed-Daten (stabil): Basis-Konfigurationen und Referenzwerte pro Umgebung
- Szenario-Daten (dynamisch): Daten, die just-in-time über APIs oder TDM-Services direkt vor dem Testlauf erzeugt oder reserviert werden
- Cleanup: Automatische Bereinigung oder Isolation ermöglicht erst die parallele Ausführung von Tests

Im Praxisbeispiel war nicht das Testskript selbst das Hauptproblem, sondern die Datenabhängigkeit. Mehrere Teams griffen auf dieselben Geschäftspartner, Verträge und Benutzer zu. Sobald parallel getestet wurde, entstanden Seiteneffekte. Erst mit reservierbaren Testdaten und klarer Isolation sank die Flaky-Rate spürbar. Der Effekt war messbar: Die Zahl der Testabbrüche aufgrund inkonsistenter Daten ging innerhalb weniger Wochen von einem regelmäßigen Problem zu einem seltenen Ausnahmefall zurück.

SIGS.DE

CI/CD: Qualitäts-Gates als Entscheidungshilfe

Automatisierung liefert nur dann Wert, wenn sie den Release-Prozess steuert. Eine moderne Pipeline unterteilt Tests in Stages mit klaren Gates, siehe Tabelle 1.

Tabelle 1: Stufen eine modernen Pileline

Stage	Fokus	Ziel-Laufzeit	Beispiel
A	Unit, statische Checks, API-Smoke	< 5 Min	JUnit
B	API/Contract & Business-Logik	< 20 Min	Pact, Postman
C	Kritische E2E-Journeys	< 90 Min	Playwright, Tricentis Tosca, UiPath
D	Performance & Security	nightly / weekly	NeoLoad, Octoperv

Im Praxisfall war genau diese Trennung der Wendepunkt. Früher lief fast alles in einem Block. Dadurch waren Fehler schwer einzuordnen, und jede Störung fühlte sich kritisch an. Nach der Umstellung wurde klar: Ein Fehler in Stage A ist etwas anderes als ein Problem in einem nächtlichen Performance-Lauf. Die Pipeline wurde damit nicht nur schneller, sondern auch deutlich besser steuerbar.

Ownership-Regel: Jeder Test braucht einen Owner. Tests ohne Owner verwaisen, werden instabil und vergiften das Vertrauen in die Pipeline.

Flaky Tests: systematisches Gift neutralisieren

„Flakiness“ ist kein Pech, sondern ein Defizit. Die Ursachen sind meist:

- Timing und Asynchronität: Stabile Tests nutzen zustandsbasierte Waits statt statischer sleep()-Befehle.
- Umgebungsrauschen: Instabile Netzwerke oder überlastete Backends.
- Abhängigkeiten: Externe Drittsysteme sollten über Service Virtualization, Mocks oder Stubs entkoppelt werden.

Ein professionelles Management identifiziert wöchentlich die „Top 10“-Übeltäter. Diese werden entweder sofort repariert oder in eine Quarantäne-Zone verschoben, um das Pipeline-Signal nicht zu verfälschen.

Auch hier helfen Metriken: In einem Team genügte bereits die Einführung eines wöchentlichen Flaky-Reports, um das Thema sichtbar zu machen. Allein dadurch sank die Akzeptanz für „den Test einfach nochmals laufen lassen“. Das ist kulturell entscheidend. Denn Flaky Tests werden nicht kleiner, wenn man sie ignoriert – sie normalisieren Misstrauen.

Abb. 1: Flaky Tests

Metriken, die das Management wirklich interessieren

Code-Coverage allein ist eine Vanity Metric. Aussagekräftiger für den Erfolg sind:

- False-Positive-Rate: Pipeline-Stopps ohne echten Produktfehler
- Time-to-Green: Zeit bis zur Wiederherstellung eines stabilen Zustands nach einem Fehler
- Business Risk Coverage: Welcher Anteil der kritischen Geschäftsprozesse ist abgesichert?
- Defect Escape Rate: Fehler, die trotz Automatisierung in Produktion gelangen

Gerade Vorher-Nachher-Metriken machen Fortschritt sichtbar. Im beschriebenen Praxisfall waren für das Management nicht 1.200 automatisierte Tests relevant, sondern drei Entwicklungen:

- die deutlich kürzere Pipeline-Laufzeit,
- die geringere Zahl an Fehlalarmen,
- die höhere Verlässlichkeit der Release-Entscheidung.

Genau dort entsteht Vertrauen.

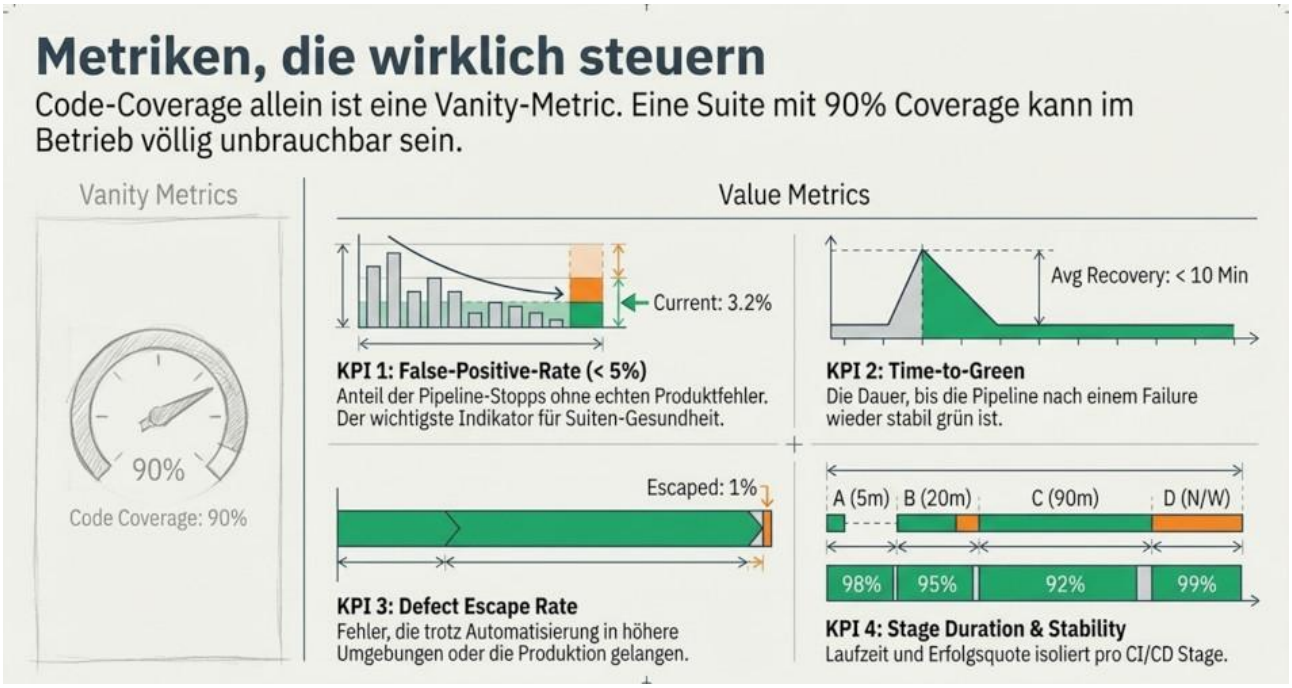

Abb. 2: Metriken

ROI: Wann rechnet sich der Aufwand?

Der Return on Investment (ROI) entsteht nicht durch die Einsparung von Personal. Er entsteht durch:

- Reduktion von Rework: Fehler im Sprint zu finden, ist deutlich günstiger als nach dem Release.
- Höhere Release-Frequenz: Von monatlichen zu häufigeren Releases kommt man nur mit belastbaren Vertrauensankern.
- Investitionsschutz: Stabilere Architekturen und saubere Abstraktion senken langfristig die Wartungskosten.
- Weniger operative Reibung: Teams verlieren weniger Zeit durch Fehlersuche, instabile Testläufe und Diskussionen über unklare Signale.

In der Praxis ist genau das oft der eigentliche Nutzen: nicht weniger Menschen, sondern mehr wirksame Kapazität.

Fazit: Automatisierung als Capability im Quality Tree

Erfolgreiche Testautomatisierung im Enterprise-Kontext ist kein „Entweder-oder“ zwischen Open Source und kommerziellen Plattformen. Die Skalierung ohne Wartungshölle gelingt durch die intelligente Kombination: Shift Left mit schlanken Code-Frameworks für die Entwicklung und modellbasierte Absicherung für komplexe Business-Journeys.

Der entscheidende Unterschied liegt nicht in der Zahl der Tests, sondern in der Architektur des Gesamtsystems. Das zeigt auch die Praxis: Erst als im beschriebenen Beispiel UI-Last reduziert, Testdaten stabilisiert, Qualitäts-Gates

getrennt und Ownership geklärt wurden, entstand aus einer teuren Testsammlung ein belastbares Automation System.

Genau hier setzt auch das Quality Tree Framework an: Testautomatisierung nicht als Sammlung einzelner Maßnahmen zu betrachten, sondern als systematischen Aufbau von Qualitätsfähigkeiten mit klaren Abhängigkeiten und echtem operativem Nutzen.

Die Zukunft gehört daher nicht den größten Test-Suiten, sondern den klügsten Automatisierungssystemen. Das Quality Tree Framework ist dazu auch als Buch im Springer Vieweg Verlag erschienen.