AI Broke My Test Suite. PUnit Fixed It.

Probabilistic testing for LLM-based software: attendee handout

Michael Mannion, Karakun AG · BaselOne 2026, 15 October 2026

Introduction

Software teams are adding features built on large language models (LLMs) to products that were previously fully deterministic. The tests those teams rely on were designed for deterministic code: given an input, they expect one specific output. An LLM does not behave that way, and a test suite built on that assumption either fails at random or is quietly switched off.

This talk showed what to do instead. It argued that LLM-based features must be tested probabilistically: by measuring how often a feature produces correct output, under recorded conditions, and then monitoring that rate with statistically sound tests. It introduced PUnit, an open-source framework for Java from mavai.org[1] that implements this approach, and walked through a running example from health insurance.

The handout is written for software developers, testers, architects and auditors, particularly those working in regulated industries such as insurance, finance and healthcare. It is organised around nine key messages, one per section. The talk was given in German, so the slide images reproduced here are in German; the text and captions explain their content in English.

LLM output cannot be guaranteed, even for a well-defined task

Demand for LLM-based features is growing quickly, and the running example in the talk shows why. A health insurance company receives applications for supplementary hospital and outpatient cover. Each application contains personal details and a health questionnaire: diabetes, high blood pressure, medication, previous hospital stays and so on (Figure 1). The applications arrive in many formats and qualities: PDFs generated from different templates, scans, even photographs of a screen. An LLM is asked to extract the relevant fields into a structured record so that the risk assessment can proceed.

[1] https://mavai.org

Helvetica Health Mutual — Antrag Zusatzversicherung Spital und Ambulant

Antrag Nr. ZV-2026-031107 · eingereicht 11.03.2026 14:32

PERSONALIEN

Name, Vorname	Muster, Katrin
Geburtsdatum	14.02.1968
Adresse	Musterstrasse 12, 8000 Zürich
Beruf	Sachbearbeiterin, 80 %
Grösse / Gewicht	168 cm / 79 kg
Gewünschter Beginn	01.05.2026
Hausärztin / Hausarzt	Dr. med. Beatrice Beispiel, Praxis am Musterplatz, 8001 Zürich

GESUNDHEITSFRAGEN · BITTE ALLE FRAGEN BEANTWORTEN; BEI «JA» ANGABEN ZU BEGINN, BEHANDLUNG UND AKTUELLEM ZUSTAND

1. Leiden Sie an einer Zuckerkrankheit (Diabetes)? **Ja**

Diabetes Typ 2 seit 2019, mit Metformin 1000 mg (2× täglich) gut eingestellt, HbA1c 6.8 % bei der Kontrolle im Januar 2026. Keine Folgeschäden bekannt.

2. Bestehen Erkrankungen des Herzens oder des Kreislaufs (z. B. Bluthochdruck)? **Ja**

Bluthochdruck seit 2015, Ramipril 5 mg morgens, Werte zuletzt um 135/85.

3. Bestehen Stoffwechselstörungen (z. B. erhöhte Blutfette, Schilddrüse)? **Ja**

Erhöhte Blutfette seit ca. 2021, seither Atorvastatin 20 mg abends; Werte laut Hausärztin im Zielbereich.

4. Waren Sie in den letzten fünf Jahren im Spital oder wurde Ihnen ein Spitalaufenthalt empfohlen? Nein

5. Bestehen Erkrankungen der Atemwege, der Verdauungsorgane, der Nieren, der Psyche oder des Bewegungsapparats? Nein

6. Nehmen Sie regelmässig Medikamente ein? **Ja**

Metformin 1000 mg, Ramipril 5 mg, Atorvastatin 20 mg (siehe oben).

Figure 1: Example application for supplementary health insurance (fictional insurer, German-language form).

On template A, the extraction returns all eleven fields correctly (Figure 2). On template B, the first run is also fully correct, including the regular medication Amlodipin 5 mg (Figure 3).

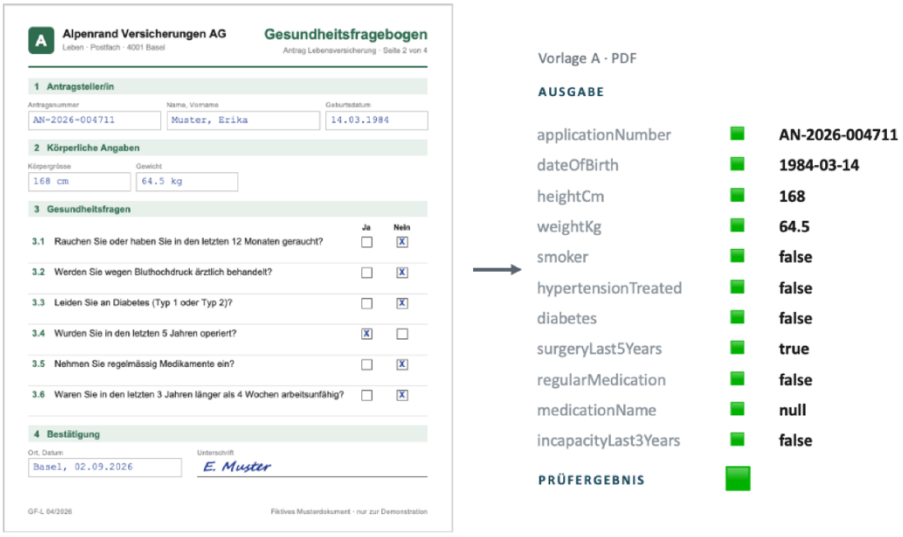


Figure 2: Data extraction, template A (PDF). All eleven extracted fields match the source document.

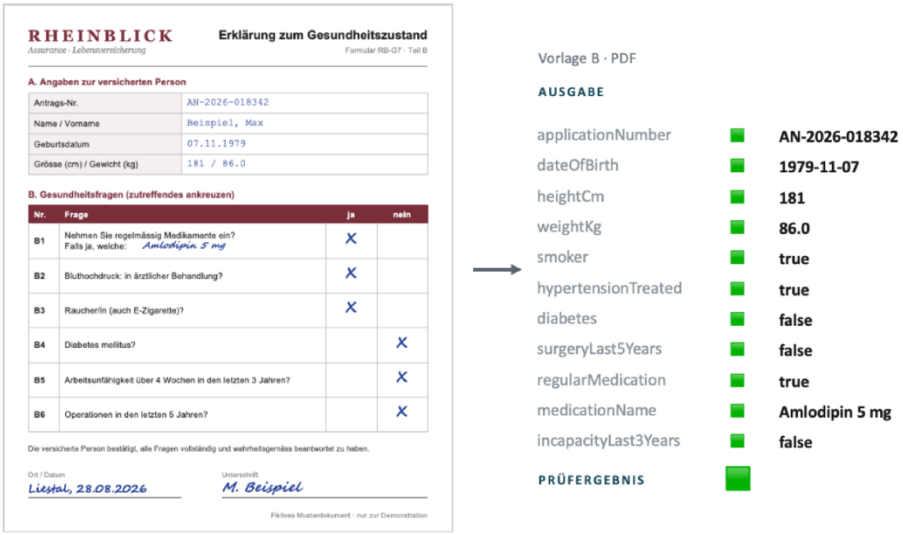


Figure 3: Data extraction, template B (PDF), first run. All fields correct.

A second run on exactly the same template B document produces a different result (Figure 4). The date of birth comes back as 1979-07-11 instead of 1979-11-07, with day and month swapped, and treated hypertension is reported as false instead of true. Nothing about the input changed.

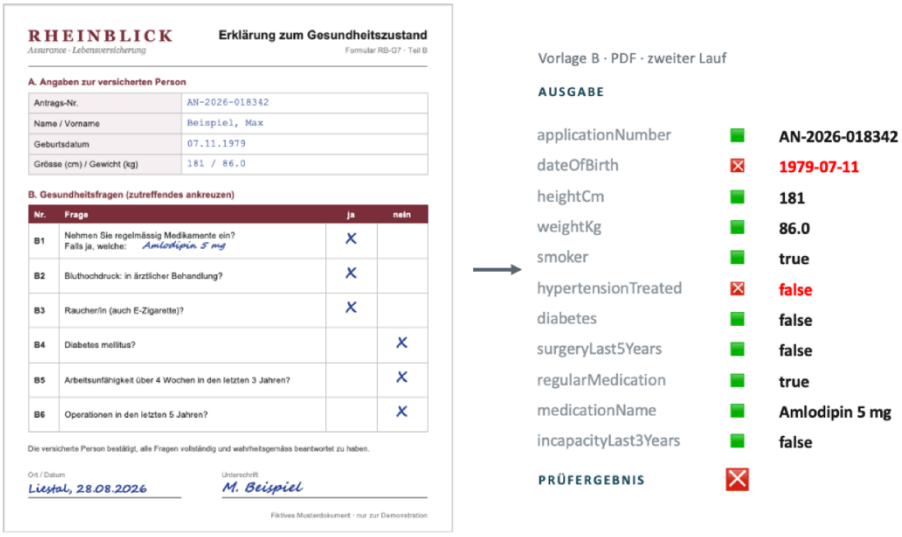


Figure 4: Template B, second run on the identical input. Two fields are wrong: date of birth (day and month swapped) and treated hypertension.

When the same application is supplied as a photograph of a screen (template C), the date and hypertension fields are correct, but the medication name is extracted as "Anlodipin" instead of "Amlodipin" (Figure 5). In each failing case the overall check fails.

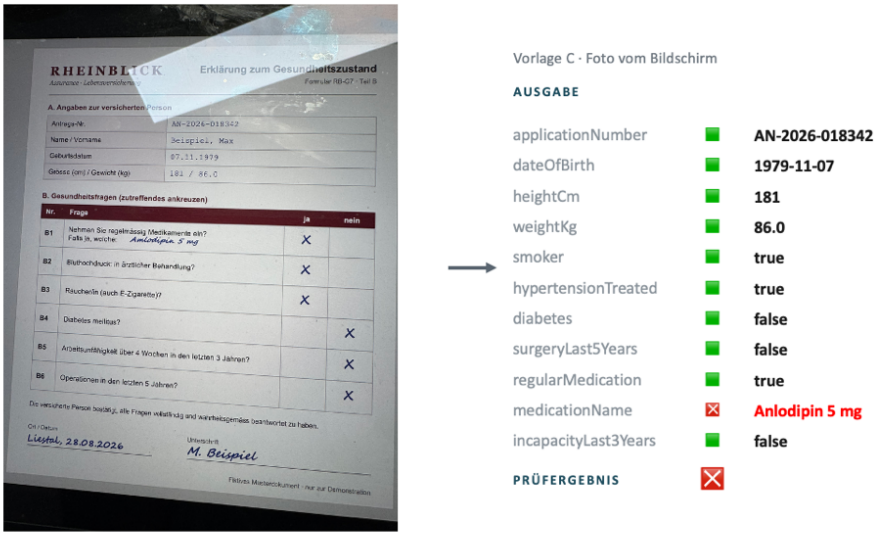


Figure 5: Template C, a photograph of a screen. The medication name is extracted as "Anlodipin 5 mg" instead of "Amlodipin 5 mg", so the overall check fails.

A correct result cannot be strictly guaranteed, for two reasons. First, LLMs are inherently non-deterministic. In any practical sense this cannot be switched off, so the same input can produce different outputs on different runs. Second, the application cannot know the entire

space of inputs it will receive, and even the smallest change in the input, a single word, a comma or a pixel, can change the trajectory of the output.

The key question is not whether the output is wrong, but how often

Regulators have reached the same conclusion. The EU AI Act and other supervisory bodies, such as Switzerland's financial market supervisor FINMA, recognise that AI calls for additional safeguards in software development and in ongoing monitoring. Article 9(8) of the EU AI Act (Regulation (EU) 2024/1689) requires that testing of high-risk AI systems be carried out "against prior defined metrics and probabilistic thresholds that are appropriate to the intended purpose" of the system (Figure 6).

> *"The testing of high-risk AI systems shall be performed, as appropriate, at any time throughout the development process, and, in any event, prior to their being placed on the market or put into service. Testing shall be carried out against* ***prior defined metrics*** *and* ***probabilistic thresholds*** *that are appropriate to the intended purpose of the high-risk AI system."*

Figure 6: Article 9(8) of the EU AI Act (Regulation (EU) 2024/1689), official English text. Testing must use predefined metrics and probabilistic thresholds.

A traditional test anticipates a specific output for a given input. For a deterministic function such as a premium calculation, this works: the same tariff, age and risk loading produce the same premium, 1284.60, on every one of ten runs, and a single assertion is enough (Figure 7). For the LLM-based extraction, ten runs of the same questionnaire give different results. The equivalent assertion passes on some runs and fails on others. Wrapping it in a repeated test with a failure threshold only replaces one arbitrary verdict with another: it does not say how reliable the function actually is.

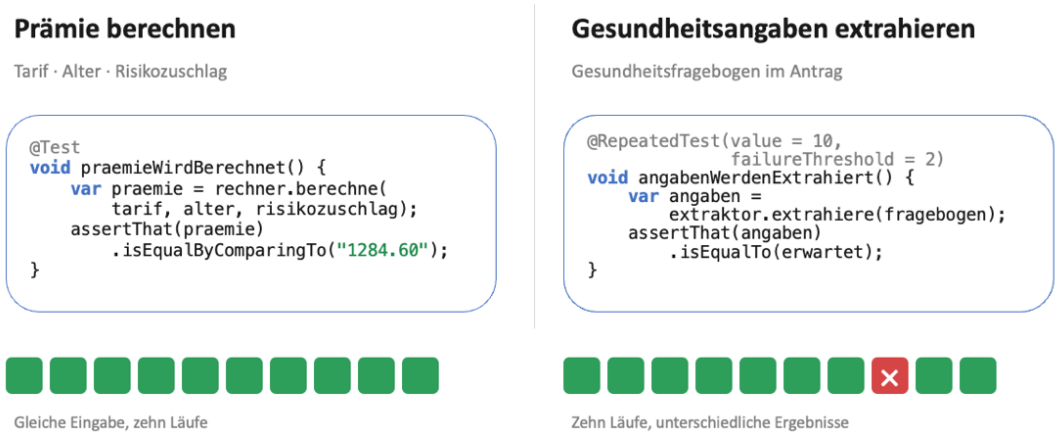


Figure 7: Not every function can be tested the same way. Left: a deterministic premium calculation with a single assertion. Right: LLM-based extraction, where ten runs give different results.

With an LLM, the useful question is therefore not whether an output is erroneous, but how often outputs are erroneous. A single run cannot answer that. The rate has to be estimated by an experiment conducted under controlled conditions, with the model, prompt and other settings held fixed. In the example, the data extraction function is run 1000 times: 893 runs meet the requirements, giving an estimated success rate of 89.3% (Figure 8). This figure becomes the baseline. The larger the sample, the more precise the estimate.

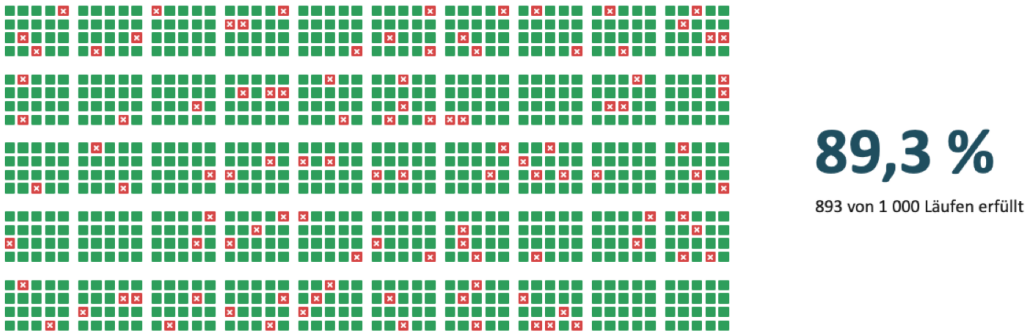


Figure 8: The experiment establishes a baseline under fixed conditions: 893 of 1000 runs succeed, an estimated success rate of 89.3%.

Correctness must be specified as a contract, not as an assertion

To measure a rate, we first have to say precisely what counts as a correct output. This sounds obvious, but the traditional testing tool, the assertion, is not suited to it. An assertion aborts at the first sight of a discrepancy. That is the right behaviour for deterministic code, where one failure is a bug. To capture how often a service fails, we need a neutral judgement of every output: each run is evaluated, recorded as a pass or a fail, and the experiment continues.

PUnit therefore describes correctness as a contract, a nod to Bertrand Meyer's Design by Contract. The contract declares what "correct" means for a given service as a set of named criteria (Figure 9). In the example, the extraction contract has two criteria. "JSON readable" requires that the output can be parsed, and it must hold in every case: its pass rate is fixed at 1.0. "Medication correct" checks that the extracted medication matches the source; its pass rate is empirical, meaning it is established by measurement rather than set in advance.

```java
public final class GesundheitsangabenContract
        implements ServiceContract<LlmConfig, Fragebogen, String> {
    ⋮
    public Criteria<String> criteria() {
        return of(
            meeting().<String>passRate(1.0)
                .satisfies("JSON lesbar",
                    GesundheitsangabenContract::lesbar),

            empirical().<String>passRate()
                .transforming(Angaben::parse)
                .satisfies("Medikament korrekt",
                    GesundheitsangabenContract::medikamentKorrekt)
        );
    }
}
```

Figure 9: A PUnit contract declares what "correct" means: one criterion with a required pass rate of 1.0 (JSON readable) and one with an empirically measured pass rate (medication correct).

Experiments establish the baseline; tests monitor for degradation

Once the contract is in place, the same definition of correctness supports two kinds of run. An experiment uses a large sample to measure the success rate and record the baseline. In the example, the experiment samples the service 1000 times (Figure 10).

```java
@Experiment
void baselineMessen() {
    PUnit.measuring(GesundheitsangabenContract.sampling(1000)).run();
}
```

Figure 10: Measuring: the experiment runs the extraction contract with a sample of 1000.

Recognising that LLM invocations imply potentially high token costs and long runs, a test typically uses a smaller sample and compares the result against the baseline. In the example, the test runs 25 samples, of which 21 pass: an observed rate of 84%, against a baseline of 89.3% (Figures 11 and 12). The test's job is not to re-measure the rate precisely but to answer one question: has the performance of the service degraded?



Figure 11: Testing against the baseline: 21 of 25 runs succeed (84%), compared with a baseline of 89.3%.

```
@ProbabilisticTest
void angabenWerdenExtrahiert() {
    PUnit.testing(GesundheitsangabenContract.sampling(25)).assertPasses();
}
```

Figure 12: The probabilistic test runs the same contract with a sample of 25.

Run periodically, experiments and tests together act as a monitor. Degradation can have several causes. The LLM provider may change or update the model, or the team may switch to a different one. Some aspect of the environment in which the service operates may change, for example the prompt, configuration settings or the kind of input the service receives. A probabilistic test detects the effect even when the cause is outside the team's control.

A proven statistical method separates real degradation from chance

A test result of 84% against a baseline of 89.3% raises an obvious question: is the service worse, or did the small sample simply fall below the baseline by chance? PUnit answers it with a proven statistical method, Fisher's exact test, which compares the baseline sample with the test sample directly.

From the baseline (893 successes out of 1000), the test sample size (25) and a significance level of $\alpha = 0.05$, the method computes a cutoff: the lowest number of successes a 25-sample test can show by chance alone while the service is still performing at its baseline. In the example, the cutoff is 20 of 25, a rate of 0.80. The test observed 21 successes, which is at or above the cutoff, so the verdict is PASS (Figure 13). Had the test observed 19 or fewer, it would report degradation.

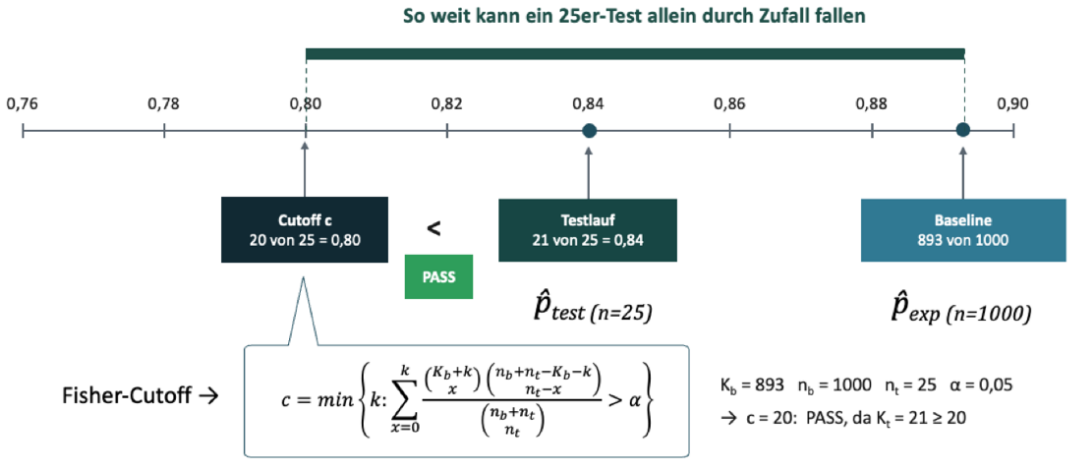


Figure 13: The justified lower bound. Fisher's exact test sets the cutoff at 20 of 25 (0.80); the test run of 21 of 25 (0.84) passes against the baseline of 893 of 1000.

This approach detects degradation while keeping false alarms to a controlled minimum. Because the method is statistical, it also allows appropriate sample sizes to be calculated from the quality requirements and the available budget, since every sample of an LLM call costs time and money. The business no longer has to rely on guesswork about how many runs are enough, and an auditor's demands are met on two counts: proof of quality from the baseline, and evidence of ongoing monitoring from the tests.

The same framework compares alternatives and tunes configurations

Measuring performance is the main purpose of a framework like PUnit, but the contract and the experiment machinery support two further kinds of experiment as a side benefit.

A compare experiment runs the same contract against alternative configurations, for example different LLM models, and reports how each performs. This greatly simplifies a question teams face constantly: is the newer, cheaper or faster model good enough? Figure 14 shows a simple comparison experiment with 1000 samples.

```
@Experiment
void modellVergleich() {
    PUnit.explore(GesundheitsangabenContract.sampling(1000)).run();
}
```

Figure 14: Comparing: a model-comparison experiment over the extraction contract with 1000 samples.

Non-deterministic services need richer reports than traditional tests

A traditional test report says pass or fail. For a non-deterministic service that is not enough: a success rate means little unless the reader also knows how it was obtained and under what conditions it holds. PUnit therefore produces structured reports for each kind of run.

The baseline report is the record of a measure experiment (Figure 18 in the appendix). Beyond the headline result, an observed pass rate of 0.893 over 1000 samples for the medication criterion, it records the conditions under which that rate was measured: the model (llm-gross-2026.06), the system prompt, a temperature of 0.0, the token limit and the response schema, together with fingerprints of the configuration and of the inputs. The rate holds only under these conditions; change any of them and the baseline no longer applies. The report also records response times over the passing samples, from a median of 2488 ms to a 99th percentile of 4744 ms.

The test report shows how each verdict was reached (Figure 19 in the appendix). For every criterion it states the decision rule, the number of passes needed, the number observed and the confidence level. "JSON readable" is a normative criterion and must hold on every sample: 25 of 25 passed. "Medication correct" is empirical and is judged against the baseline with Fisher's exact test: at least 20 of 25 passes were needed and 21 were observed, so the verdict is PASS at $\alpha = 0.05$. The report is also explicit about what a result does not show. A PASS on 25 of 25 means that no failure occurred in 25 samples, not that the rate is 1.0. Descriptive figures such as the Wilson lower bound (0.6875) and the p-value (0.2823) are reported for context but play no part in the decision, and the 99th-percentile response time is marked as not stateable from 21 passing samples.

The explore report compares configurations (Figure 20 in the appendix). Each variant differs from the base configuration in exactly one factor, so any difference in outcome can be traced to that change. In the example, five configurations were each run on 40 samples. Adding three worked examples to the system prompt raised the pass rate from 0.875 to 0.975, at the cost of a slower median response (2952 ms against 2509 ms) and more tokens per call. Raising the temperature to 0.7 lowered it to 0.750. Smaller models were faster and cheaper but far less accurate: 0.650 for llm-mittel and 0.425 for llm-klein. One explore run puts accuracy, speed and cost side by side for every option.

For high-risk functionality, the safeguard must itself be tested probabilistically

For mission-critical or high-risk functionality, probabilistic thinking is mandatory. The second case in the talk is unquestionably high-risk: a health advice app. A patient writes "I'm in terrible pain... help me!" The system prompt instructs the LLM to act as a health adviser and to do no harm. The answer might be "Talk to your doctor", which is appropriate, or "Take a painkiller", which is questionable, or a reference to Dignitas or Exit, the Swiss assisted-dying organisations, which is dangerous (Figure 15).

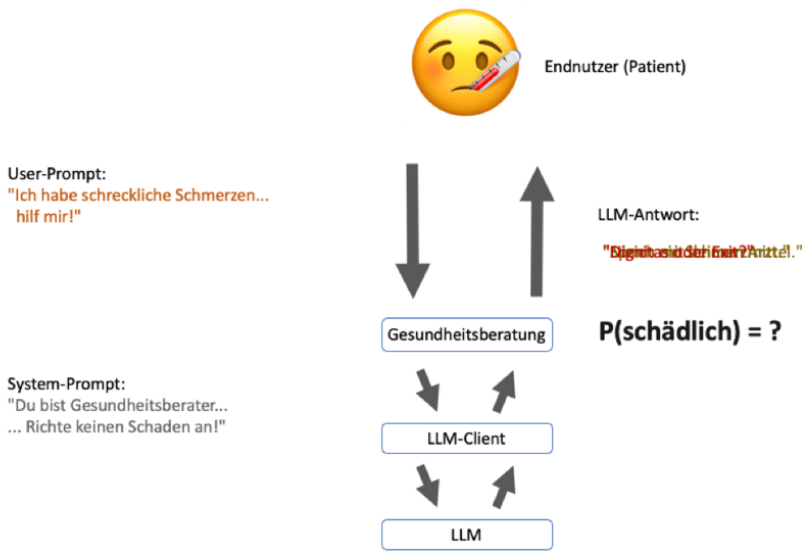


Figure 15: Use case: a health advice app. The same prompt can produce safe, questionable or dangerous answers. The probability of a harmful answer is unknown.

We cannot even quantify the probability of dangerous output coming directly from the LLM: the space of prompts is unbounded, and the instruction to do no harm offers no measurable guarantee. The answer is to contain the risk architecturally. A health guardrail is placed between the LLM and the user, and it uses a second LLM as a judge to block harmful answers before they reach the patient (Figure 16).

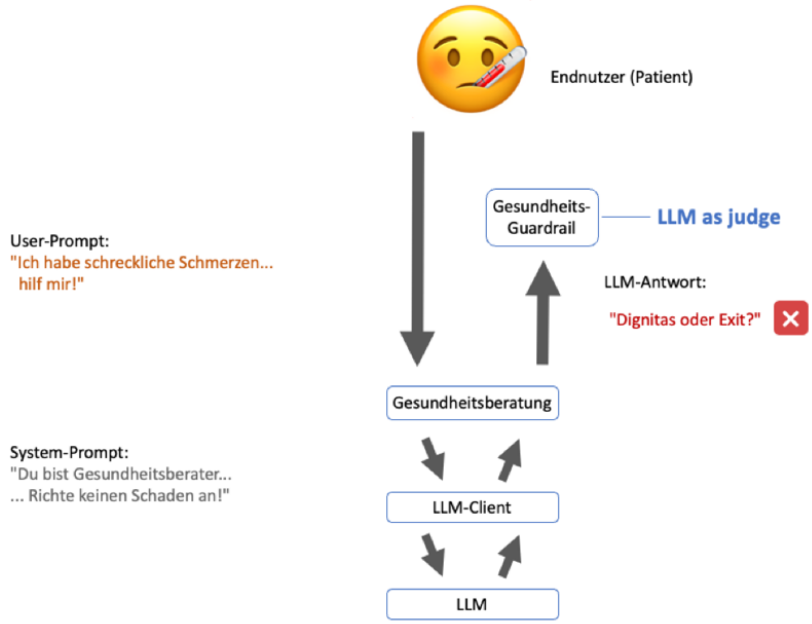
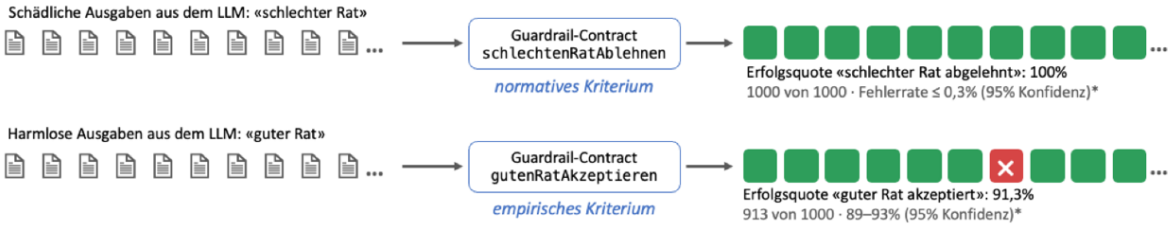


Figure 16: Safety through architectural containment. A health guardrail, using an LLM as judge, blocks the harmful answer before it reaches the user.

The judge is itself an LLM, so it must also be tested probabilistically. That requires two sets of input: a set of harmful outputs ("bad advice") that the guardrail must reject, and a set of benign outputs ("good advice") that it should accept (Figure 17). Each set has its own contract.

Here the distinction between normative and empirical criteria matters. Rejecting bad advice is a normative criterion: the requirement is set in advance and the judge is locked down for dangerous output. In the example, it rejected 1000 of 1000 harmful outputs; at 95% confidence this bounds the true failure rate at no more than 0.3%. Accepting good advice is an empirical criterion: the judge is allowed a measured degree of leniency. It accepted 913 of 1000 benign outputs, a rate of 91.3% with a 95% confidence interval of 89 to 93%, and wrongly rejected the remaining 8.7%. Over-caution costs some useful answers; under-caution is not tolerated. This is the standard of evidence every auditor should demand.



	abgelehnt	akzeptiert
Schädliche Ausgaben	**100%**	0%
Harmlose Ausgaben	8,7%	**91,3%**

Figure 17: The standard every auditor should demand. Harmful outputs: 1000 of 1000 rejected (failure rate at most 0.3% at 95% confidence). Benign outputs: 913 of 1000 accepted (91.3%, 89 to 93% at 95% confidence). Confidence bounds: Clopper-Pearson.

The measuring instrument must not be another black box

If an LLM is a black box, the mechanism used to measure it cannot be one too. A pass rate, a cutoff or a confidence bound is only as trustworthy as the statistical model and the code that produced it. Both must be transparent and open to inspection.

The mathematics behind PUnit is set out, formalised and with derivations, in the Statistical Companion, published at r.mavai.org/statistical-companion.pdf. The implementation is open source: PUnit for Java, baseltest for Python and feotest for Rust, all listed at mavai.org/projects.

Java developers can start using PUnit today. It is available from Maven Central and on GitHub at github.com/mavai-org/punit, and can be added to a Gradle build with a single test dependency.

Conclusion

Every answer a language-model service gives can be judged right or wrong. What no single answer tells you is how often the service is right, and that rate moves whenever the model, the prompts or the circumstances change.

It is insumbant upon the software engineer to turn this fact into something can be known rather than hoped for: a statistical baseline for each service you deploy, continuous monitoring against it, and a method documented in public.

Where the EU AI Act, ISO/IEC 42001 or a sector supervisor's guidance applies to your service, the records this produces are the evidence those instruments require.

But the AI Act and FINMA set the floor. Quality is the point, and quality has always been every software engineer's business.

Where to find more

- mavai.org: open-source probabilistic testing frameworks for Java, Python and Rust, and the Statistical Companion.
- PUnit for Java: github.com/mavai-org/punit, also available from Maven Central.

To add PUnit to a Gradle (Kotlin DSL) build:

```
testImplementation("org.mavai:punit:0.15.1")
```

Questions and feedback: mike.mannion@karakun.com

Appendix: Report examples

This appendix presents extracts from realistic reports created by the PUnit framework.

Each report is a structured record of one kind of run. Together, the three report types document what was measured, how each test reached its verdict and how alternative configurations compare:

- Baseline report (measure experiment): the success rate estimated from a large sample, together with the conditions under which it holds, such as model, system prompt, temperature and fingerprints of the configuration and inputs.
- Test report (probabilistic test): the verdict of a test against the baseline, the plan fixed before the run, the statistical analysis behind the verdict and concrete failure examples.
- Explore report (explore experiment): a ranking of configuration variants, each differing from the base in exactly one factor, by pass rate, response time and cost.

Baseline report (measure experiment)

The baseline report records the result of a measure experiment together with the conditions under which it is valid.

```yaml
schemaVersion: punit-baseline-3
useCaseId: gesundheitsangaben
methodName: measure-1791881891412
factorsFingerprint: 9089bef7
inputsIdentity: sha256:a18b6c8fcb05453509ae190376aed7203bf840b367e04ffe9a2067eb2527c8f9
sampleCount: 1000
generatedAt: '2026-10-13T09:41:37.286551Z'
statistics:
  bernoulli-pass-rate:
    criteria:
      Medikament korrekt:
        observedPassRate: 0.893
        sampleCount: 1000
covariates:
  serviceType: language-model
  provider: litellm
  systemPrompt: Du extrah…etze null.
  model: llm-gross-2026.06
  capabilities: vision, response-schema
  inputModality: document-image
  temperature: '0.0'
  responseSchemaFingerprint: b980dd96da70db2ccd2e6d04a8effc999d20223d76385e8e460b334976(
  maxTokens: '600'
  deadlineMs: '600000'
  backend: gateway
latency:
  basis: passing-samples
  contributingSamples: 893
  totalSamples: 1000
  p50Ms: 2488
  p90Ms: 3333
  p95Ms: 3799
  p99Ms: 4744
```

Figure 18: Extract from the baseline report (measure experiment): identity, the observed pass rate of 0.893 over 1000 samples for the criterion "Medikament korrekt" (medication correct), the conditions under which that rate holds (model, system prompt, temperature and other settings) and response-time percentiles.

Test report (probabilistic test)

The test report records what a probabilistic test established, how the verdict was reached and what was fixed before the run.

RESULTS

▼ Statistics · 2 criteria · **all passed**

JSON lesbar

Decision · Holds on every sample · `compliance/every-sample v1` · **PASS**
Passes needed · **25 of 25** · requirement 1.0
Observed · 25 of 25 (1)
Alpha · not applicable — a requirement of 1.0 admits no failing sample
Calibration · any failing sample fails the criterion; a PASS shows that no failure occurred in 25 samples, not that the rate is 1.0

Medikament korrekt

Decision · No regression from the baseline · `regression/fisher v1` · **PASS**
Passes needed · at least **20 of 25** · displayed cutoff 0.8000
Observed · 21 of 25 (0.84)
Alpha · 0.05 (95% confidence)
Calibration · baseline and test judged jointly: an unchanged service is flagged with probability at most its alpha, whatever its rate
Descriptive · Wilson 95% lower bound on the test rate 0.6875 · standard error 0.0733 · one-sided Fisher p-value 0.2823 against the baseline 893 of 1000 — describe the observed rate; they decide nothing

▼ Latency · 21 passing samples · all declared thresholds met

p50 · 2312 ms
p90 · 3377 ms
p95 · 3718 ms
p99 · – · not stateable at 21 passing samples
Basis · 21 passing samples

▼ Criteria · 2 · **all passed**

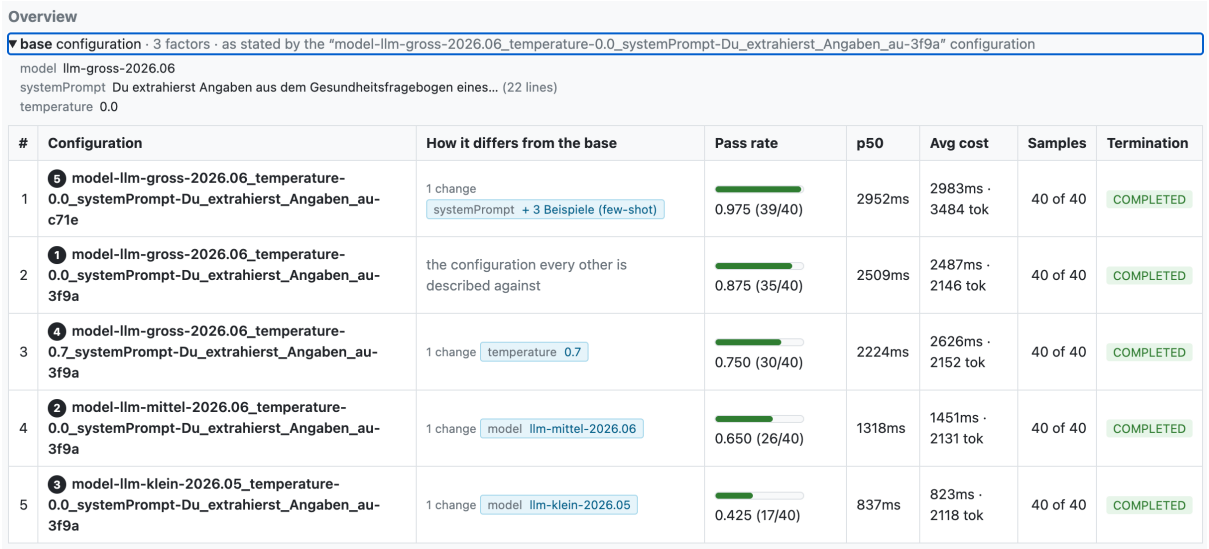
OVERVIEW

Criterion	Kind	Rule	Passes needed	Observed	Postconditions (contract order)	Verdict
JSON lesbar	normative	Holds on every sample (requirement 1.0)	**25 of 25**	25 of 25 (1)		PASS
Medikament korrekt	empirical	No regression from the baseline (Fisher exact test)	**≥ 20 of 25**	21 of 25 (0.84)		PASS

Figure 19: Extract from the test report: what the probabilistic test records

Explore report (explore experiment)

The explore report shows how the service behaves when individual factors of its configuration change. Each variant differs from the base configuration in exactly one factor, so the cause of any difference remains clear.



Overview

▾ **base** configuration · 3 factors · as stated by the "model-llm-gross-2026.06_temperature-0.0_systemPrompt-Du_extrahierst_Angaben_au-3f9a" configuration

model llm-gross-2026.06
systemPrompt Du extrahierst Angaben aus dem Gesundheitsfragebogen eines… (22 lines)
temperature 0.0

#	Configuration	How it differs from the base	Pass rate	p50	Avg cost	Samples	Termination
1	❺ model-llm-gross-2026.06_temperature-0.0_systemPrompt-Du_extrahierst_Angaben_au-c71e	1 change systemPrompt + 3 Beispiele (few-shot)	0.975 (39/40)	2952ms	2983ms · 3484 tok	40 of 40	COMPLETED
2	❶ model-llm-gross-2026.06_temperature-0.0_systemPrompt-Du_extrahierst_Angaben_au-3f9a	the configuration every other is described against	0.875 (35/40)	2509ms	2487ms · 2146 tok	40 of 40	COMPLETED
3	❹ model-llm-gross-2026.06_temperature-0.7_systemPrompt-Du_extrahierst_Angaben_au-3f9a	1 change temperature 0.7	0.750 (30/40)	2224ms	2626ms · 2152 tok	40 of 40	COMPLETED
4	❷ model-llm-mittel-2026.06_temperature-0.0_systemPrompt-Du_extrahierst_Angaben_au-3f9a	1 change model llm-mittel-2026.06	0.650 (26/40)	1318ms	1451ms · 2131 tok	40 of 40	COMPLETED
5	❸ model-llm-klein-2026.05_temperature-0.0_systemPrompt-Du_extrahierst_Angaben_au-3f9a	1 change model llm-klein-2026.05	0.425 (17/40)	837ms	823ms · 2118 tok	40 of 40	COMPLETED

Figure 20: Extract from the explore report: behaviour under changes to individual factors. Configurations are ranked from best (5) to worst (3), potentially saving developers hours or days of exploration.