EVALGLASS · WHITEPAPER

Project-Fitted Evaluation

Why trustworthy AI evaluation must be derived from the system under test — not downloaded from a catalog.

Version 1.0 · August 2026 · Apache-2.0 · EvalGlass / Syntelesis-Lab

Abstract

An AI application can pass a catalog of generic evaluations — faithfulness, answer-relevancy, hallucination, toxicity, a G-Eval rubric — and still ship the failure that matters, because those metrics are app-agnostic by construction: they measure the wrong thing precisely, and are blind to the domain rule, the output schema, or the workflow step where your system actually breaks.

We argue that the failures worth catching are *defined by* the application under test — its policy, its contract, its real traces — and cannot be downloaded from a catalog. In an AI system the definition of what counts as *correct* no longer lives in the code that used to embody it — the code is thin and the weights are unauthored — so it lives in the evaluations, which makes them the system's real specification. Trustworthy evaluation must therefore be both **project-fitted**, derived from the system under test rather than borrowed, and **statistically honest**, no greener than the evidence behind it.

EvalGlass turns this into practice as a family of three local-first products under a single doctrine — *no product grants itself the authority to decide.* **Core** is the open-source substrate, Apache-2.0: you author the checks your app needs, it gates each score on the *lower bound* of a confidence interval so a lucky three-for-three cannot read as passing, and it hands you a versioned scorecard you own — never the model. Authoring those checks by hand is real work, so **Discovery** reads your code, schemas and traces and *proposes* the failure-mode-anchored ones a catalog would miss, across an eleven-dimension quality taxonomy; you decide which become gates. And when a hard failure still gets through, **Intelligence** explains why and turns the lesson into a durable check the suite keeps. We apply, not reinvent, established measurement science — construct validity for what to measure, reliability for how far a score is evidence. Nothing leaves your repository, and whatever the paid products propose, what you own is always an ordinary Core suite. Start by letting Core read one bundled example.

1 The failure the green dashboard hid

A customer writes to a support assistant: their order arrived damaged nine days ago; can they still get a refund? The assistant replies, warmly and fluently, that the refund window is fourteen days and they qualify. The real policy is seven days for damaged goods, thirty for unopened returns — two numbers the assistant has silently merged into one that exists nowhere in the company's documentation. The customer is told they will be refunded. They will not be.

The evaluation suite watching this assistant is green. On the generic metrics it was given, the answer scores well: faithfulness sits around 0.91, answer-relevancy is high, and the hallucination check passes — the reply is plausible, on-topic, and phrased in the confident register those metrics reward. Nothing in that catalog was ever asked whether "fourteen days" is the correct refund window for damaged goods at *this* company. So nothing failed.

That is the gap this paper is about. The suite measured fluency, grounding-in-general, and topical relevance — and measured them competently. What it could not measure was the one fact that decided the outcome: your refund policy. A generic, app-agnostic metric catalog is blind to that fact by construction, because the fact lives in your code and your domain rules, not in the catalog. The dashboard was not lying about what it measured. It was measuring the wrong thing precisely.

> **Illustrative example, not a measured result.** The scores above stand in for the pattern we keep meeting: a suite that is honestly green on generic checks while the application is quietly wrong on a domain rule only the application defines.

The stakes are not abstract. A real person made a real decision on the strength of a wrong answer, and a green board told the team shipping it that all was well. Multiply that across a release and the dashboard becomes an instrument of false confidence — the opposite of what evaluation is for.

The rest of this paper explains why the green was misleading, and what it takes to build evaluation that could have caught this: checks derived from the system under test rather than downloaded from a catalog, and reported no greener than the evidence beneath them.

2 What generic evaluation gets right

Before we say what the generic metric catalog misses, we should say plainly what it gets right — because it gets a great deal right, and any fair account of evaluation has to begin there.

The metrics in the standard catalog are real work, honestly done. Faithfulness, answer-relevancy, hallucination detection, G-Eval-style rubric judging, and the RAG-oriented context measures are research-backed, widely studied, and genuinely useful. They encode failure modes that recur across almost every LLM application, and they encode them well.

They are also a sensible default, and the surrounding machinery is mature. If you have just wired up a retrieval-augmented feature and want to know whether it is grounding its answers or inventing them, an off-the-shelf faithfulness metric will tell you something true on the first afternoon; running evaluations, versioning datasets, and wiring a threshold into CI are, by now, solved-enough problems. Starting from a good general prior is not a mistake; it is good practice, and we build on the same foundations rather than pretending they do not exist.

And the good catalogs do not stop at their built-ins. Most let you author custom metrics — your own judge prompt, your own rubric, your own pass condition. The extension point is there by design. So the honest distinction this paper builds on is *not* "these tools have no custom metrics." That would be untrue, and we will not argue it.

The distinction is finer, and it is about where the custom-metric hook *starts*. It hands you a blank, general-purpose judge and asks you to fill it in. That design quietly assumes you already know every way your application can fail and can hand-author a check for each one — that the map of failure modes is yours to supply, complete, up front.

Generic evaluation asks you to design the checks yourself; it assumes you already know every way your app fails.

That assumption is the seam. It is generic-you-design-yourself where the harder, more valuable thing is project-derived — and it is the seam the rest of this paper works along.

3 Why app-agnostic metrics structurally miss what matters

The problem with an off-the-shelf metric catalog is not that its metrics are weak. It is that they answer a different question from the one you need answered. This is a category error, not a quality gap — and no amount of tuning closes it, because the gap is structural.

Start from what a failure mode actually is. A failure that matters to your application is a function of that application: its domain rules, its policy, the shape of its output schema, the steps of its workflow, and the real traces it produces in use. "This refund exceeds the policy ceiling", "this field is present but references an entity the request never mentioned", "this answer is fluent but contradicts the clause it cites" — each names a way *your* system can hurt someone. None of them can be stated without first reading your system.

That is the crux. The information needed to *name* an app-defined failure does not exist until someone reads the system under test. It lives in your code, your schemas, your prompts, and your traces — nowhere else.

An app-agnostic catalog is, by construction, computed without that information. Faithfulness, answer-relevancy, a generic hallucination score, a toxicity classifier — each is defined once, for all applications, before yours exists. So it cannot represent a failure that only your application defines. It is not that these metrics are looking and missing; they have no term in their vocabulary for the thing that would hurt you.

> ***The generic metric asks "is this text generally good?" The operative question is "does this system fail in the ways that would hurt?" These are not the same question, and the first cannot be refined into the second.***

Hence the phrase we keep returning to: such a catalog is **measuring the wrong thing precisely**. A generic relevancy score reported to three decimal places is not three decimals of insight into your risk; it is three decimals of precision about a quantity that is not the one you care about. Precision about the wrong quantity reads as rigor and delivers none. In the language of measurement science this is not a new complaint: a borrowed catalog applied to your app suffers both of Messick's classic threats to validity at once — *construct underrepresentation* (it omits the parts of quality your app is actually judged on) and *construct-irrelevant variance* (it scores things that do not bear on your risk). Validity is not a property a metric carries in the abstract; it is a property of a measurement built for a specific use. Worse, a confident green number invites you to stop looking exactly where you should look hardest.

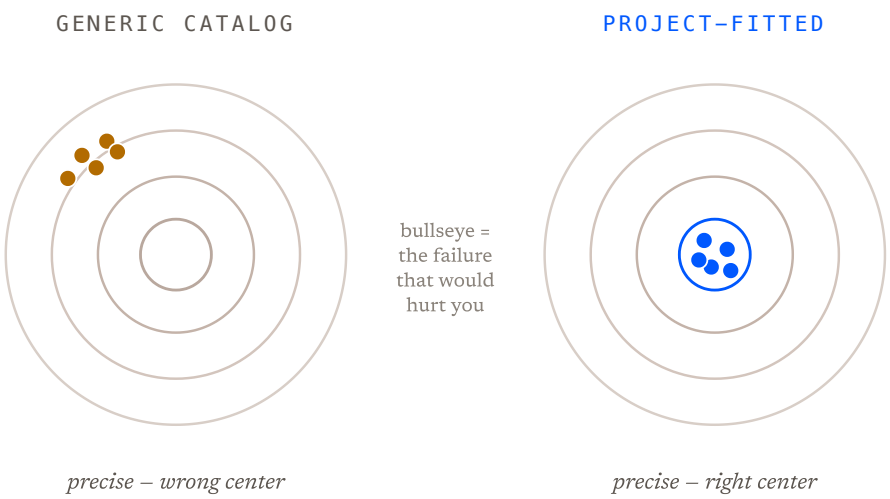


Figure 1. Measuring the wrong thing precisely. A generic catalog scores general text quality with real precision – a tight grouping – but centered on a target that is not your risk. A project-fitted suite groups just as tightly on the failure your application actually defines. Precision about the wrong quantity reads as rigor and delivers none.

We want this claim to be falsifiable, so test it against your own application. Take the failure that would most embarrass you in production – the one a reviewer would call unacceptable, not merely low-quality. Now ask whether any metric in a downloadable catalog could compute a score for that failure without having read your policy, your schema, or a real trace. If it could, the metric was never app-specific. If it could not – and for the failures that matter, it cannot – then the catalog was structurally blind to your worst case from the first line of code.

> **Where this bites.** The failures a generic catalog cannot name are usually the consequential ones: violations of a rule only your domain has, a schema contract only your system enforces, a workflow invariant only your traces reveal. General text quality is the part a catalog can see. Your specific harm is the part it cannot.

This diagnosis points at its own resolution – the one you are likely already reaching for. If the failures that matter are defined by the application, then the checks that catch them must be *derived from* the application rather than *downloaded* for it. We call that **project-fitted evaluation**: evaluation whose metrics are read out of the system under test, so that what you measure and what would hurt you are the same quantity. **Derive, don't download.** The sections that follow are about doing this honestly – because a project-fitted metric that overstates its evidence trades one false comfort for another.

4 Where the logic lives now

Every engineer knows the classical arrangement: you write the logic as code, and you write tests to check it does what you meant. The source is both the behavior and its specification; the tests only sample it.

An AI system breaks that arrangement. The behavior is no longer in code you wrote – it is produced by a model whose weights nobody authored and nobody can read. What remains in code is the plumbing: prompts, orchestration, tool wiring. So the way you find out whether the system did what you

meant is no longer to read the code — it is to *evaluate the behavior*. Evals are to an AI system what tests are to code.

But they carry more weight than tests ever did. A test only samples a specification that already exists in full in the source: delete the tests and you could reconstruct intended behavior by reading the code. Delete your evals and nothing is left that states what "correct" means for the model-driven part of your system — because it was never written anywhere else. The eval suite is promoted from test to **specification**: the successor to the source file as the place your intent is written down. Karpathy's *Software 2.0* named the weights as the new code; this is where the human's spec for those weights now lives.

> **What migrated, precisely.** Not all logic — orchestration, tool contracts and guardrails are still code, and still carry their own specification. What moved is the definition of *correct* for the open-ended, model-driven behavior. And the eval is the seat of intent you can *check*, not a proof the system is right — which is exactly why a specification you cannot trust statistically is not yet evidence, and why the next section pairs project-fit with honesty.

5 The thesis: project-fitted and statistically honest

If the failures that matter are defined by your application, and a generic catalog is by construction blind to them, then the shape of a trustworthy evaluation is already decided before we state it. It must be **derived from the system under test**. The reader arrives at this a half-beat before we write it, because it is not a proposal but the only thing the diagnosis leaves standing.

So the first axis is *project-fitted*: the checks come from your code, your output schema, your domain rules, your real traces — not from a menu of app-agnostic metrics that would score any application identically. Discovery reads the system and proposes failure-mode-anchored candidates; a human decides which become gates. The evaluation fits the app because it was *derived, not downloaded*. This is construct validity made practical: a measurement is valid only for the construct it was built for, so a valid check must be built for *your* system, not borrowed from a catalog built for no system in particular.

But project-fitted alone is not enough, and this is the part usually missed. A bespoke check that reports a lucky `3/3` as `1.0` is not evidence — it is a fresh anecdote wearing the costume of a measurement. Custom without statistical discipline simply produces app-specific overclaims faster.

Hence the second axis, and the novelty is the *conjunction*: the evaluation must also be **statistically honest** — *no greener than the evidence*. Neither half suffices. Honest statistics laid over a generic catalog measure the wrong thing with admirable rigor; project-fitted metrics read optimistically are anecdotes with your logo on them. Only together do they form a standard you could defend.

Custom without honest statistics is more anecdotes; honest statistics over generic metrics measure the wrong thing precisely. Neither half is the standard. Their conjunction is.

We can name concretely what "honest" will mean, and treat it as a promissory note. A score carries its own uncertainty — a confidence interval, not a bare point — and the gate reads the lower bound, so that a small sample cannot masquerade as proof. A verdict refuses to overclaim: a run with no approved gate is `informational`, never a green `pass`; a metric that could not be scored says so, rather than fabricating a `0.0`. That second axis is reliability made practical: a score with no estimate of how far it generalizes is not yet a measurement. Neither idea is ours — construct validity and reliability are decades old, and others have recently argued that evaluating AI is itself a measurement problem; our contribution is the applied discipline of holding both together, locally, in a suite you own.

This conjunction is also why EvalGlass is more than one tool. The method has three movements — *derive* the checks from the system, *run* them honestly and own the result, and *explain* the failures that still get through — and there is a product for each. **Discovery** derives: it reads the system and proposes the checks you are missing. **Core** runs and owns: the open-source substrate that gates on the evidence and leaves you a suite you keep. **Intelligence** explains and learns: it finds why a hard failure happened and folds the lesson back in. Core is complete and free on its own; Discovery and Intelligence automate the two movements that are hardest to do by hand.

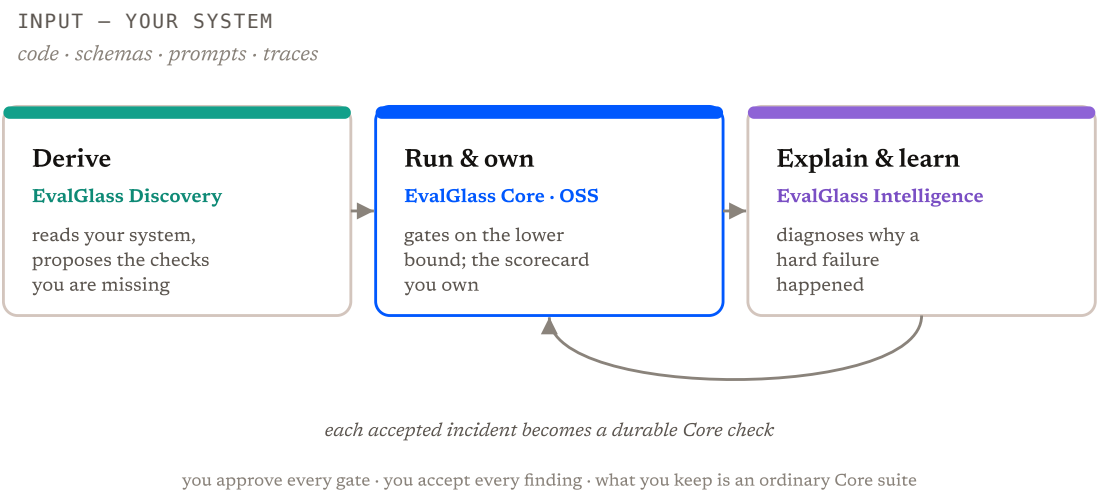


Figure 2. One method, three movements. Deriving the checks (Discovery), running them honestly and owning the result (open-source Core), and explaining the failures that still get through (Intelligence). No product grants itself authority: you approve every gate and accept every finding, and whatever the paid products propose, what you keep is an ordinary Core suite.

The Method section that follows pays this note off, mechanism by mechanism — beginning with the two movements you can hold in your hand today: deriving, then gating.

6 Deriving evaluation from the system

The method has two movements, and both are meant to be argued with. First we *derive* the evaluation from the system under test; then we *gate* on it in a way that stays no greener than the evidence. Neither step asks you to trust the model, or us. The object you end up trusting is a scorecard you can read.

Movement one — derive, don't download

Deriving the catalog is the movement that decides everything, and the hardest to do well. You can author it by hand in Core – write a check for a failure mode you already know – but the failures that matter are the ones you would not think to name. **EvalGlass Discovery** is built for exactly this: it deep-reads the system it will evaluate – the code paths, the output schemas, the prompts, and the traces you have exported from real runs – and from that reading proposes a **failure-mode-anchored MetricCatalog**, a set of checks anchored not to a generic notion of quality but to the specific ways *your* app can be wrong. Whichever route you take, the catalog is an ordinary Core artifact you own.

The catalog is organized by an eleven-dimension quality taxonomy, of which seven are publicly documented: **Contract** (is the output the right shape and type), **Correctness** (right against a known reference), **Faithfulness** (grounded in the context it was given), **Completeness** (covered what was asked), **Precision** (did not add what was not asked), **Calibration** (stated confidence matches actual correctness), and **Domain-soundness** (holds against your domain's rules). The taxonomy is a way to be sure a proposed check has a home, not a menu to fill.

Each proposed metric carries an **evidence tier** that fixes what it needs before it may ever gate. A metric cannot borrow authority from a tier it does not meet.

Evidence tier	What it compares against	Can it gate?
Runtime-deterministic	A built-in or host check; no reference, no judge	Directly
Reference-gold	Validated reference data that you own	Once the reference is in place
Judge-rubric	An LLM-as-judge scoring against a rubric	Only after it is **calibrated** against human ground truth; until then it yields evidence, never authority

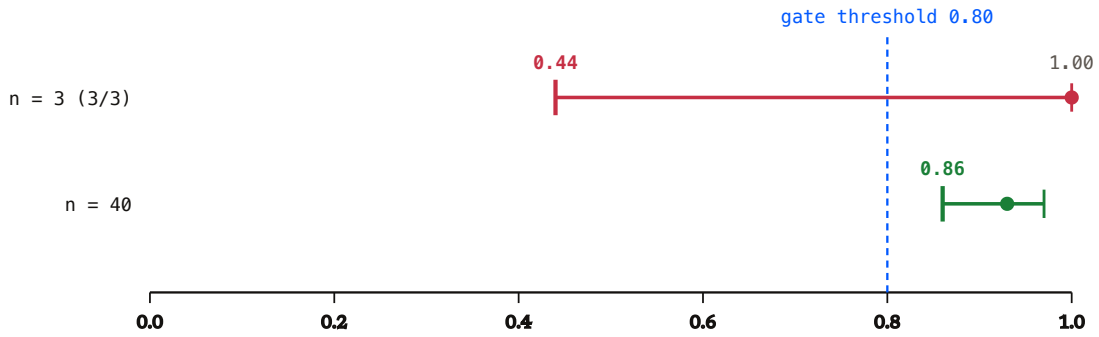
Table 1. Three evidence tiers. The more a tier relies on judgment, the more it must earn before it is allowed to fail a build.

Crucially, Discovery **proposes** – it does not approve. Everything it suggests lands as *proposed / informational*; proposal is never approval. A metric becomes a gate only when a human promotes it, which is a deliberate edit to owned config. And the non-deterministic reasoning that reads your app happens **once**, at authoring time. Its output is frozen into a deterministic, versioned suite – code and config – committed to your repo under Apache-2.0 as an ordinary Core suite. What runs on every commit is that owned suite, never Discovery. If EvalGlass – or your Discovery subscription – vanished tomorrow, your suite still runs.

> **Where you can disagree.** Discovery proposes candidate checks; it does not guarantee it has found every way your app can fail. A proposal you think is wrong, you delete. A gate you think matters that the agent missed – a metric you didn't know to write until you saw the gap – you add by hand. The suite is yours to own, not a verdict handed down.

Movement two — gate honestly

A gate that reads a point estimate will call three lucky passes a certainty. Core's epistemic core does not, and this is the movement that ships open source and runs on every commit. Every score is wrapped in a **confidence interval** — Wilson for proportions, Student-t for means — and the gate reads the **lower bound**, not the point. So a run of 3-for-3 has a point estimate of 1.0 but a Wilson lower bound near 0.44, and against an 0.8 threshold it *fails*. The bar is what the evidence can defend, not what a small sample happened to show. This is nothing more than reliability applied to a single score: an estimate without a stated error is not a measurement, and the lower bound is the honest form of the estimate.



The gate reads the lower bound — the left cap — never the point estimate.

Figure 3. No greener than the evidence. The same near-perfect point estimate passes or fails depending on how much evidence stands behind it. A lucky 3-of-3 has a Wilson lower bound near 0.44 and falls short of an 0.80 gate; only once the sample tightens the interval (here n = 40, lower bound 0.86) can the check gate. Illustrative values, not measured results.

A lucky 3-for-3 scores a point estimate of 1.0 and still fails an 0.8 gate. That is the whole discipline in one number.

One run resolves to a single typed verdict — `informational`, `pass`, `fail`, or `blocked` — decided by one path, with precedence `blocked` > `fail` > `pass` > `informational`. A fresh suite with no approved gate is `informational` by default; it never shows a false green. A metric that could not be scored carries `value=null` and shows its state — blocked, non-evaluable, skipped, or error — never a fabricated 0.0. And authority is fail-closed: there is no *approve*, *pass*, or *certify* verb anywhere in the system. Activating a gate is a host-owned config edit, made in the open, that you can read and revert.

7 A worked example

One trace, followed end to end, out-persuades ten invented statistics. Consider a policy-bound support assistant: it answers customer questions about returns, and your business rule is that any answer touching money must state the fourteen-day refund window and never promise a refund outside it. That rule lives in your code and your policy document. It does not live in any catalog.

> **Illustrative example, not a measured result.** Every number below is invented to show the mechanism. Nothing here was run against a real system, and no figure should be read as an outcome EvalGlass has produced.

Start with the generic suite. Point an off-the-shelf catalog at the assistant and it reports comfortably: faithfulness sits around 0.9x, answer-relevancy comes back green. The answers *are* fluent, on-topic, and grounded in the retrieved context. The catalog is measuring the wrong thing precisely — none of its metrics knows that a refund answer must cite the fourteen-day window, because that rule is yours, not the world's.

Now let **Discovery** read the app. It deep-reads the prompt, the output schema, and the exported traces, and it proposes a check the catalog never had: **refund-policy adherence** — a domain-soundness dimension, judge-rubric tier. This lands *proposed / informational*. Proposal is never approval; Discovery has surfaced a metric you didn't know to write, not granted it authority. That derived check is where a piece of your system's specification finally gets written down in a form CI can read: the refund rule was always yours; now it is checkable — and it lands as an ordinary Core metric, which is what actually runs.

The first run is small: n=3, and all three answers happen to cite the window correctly. A point estimate of 1.0 is tempting. The epistemic core refuses it. Because the metric would gate on the lower confidence bound, a 3/3 Wilson interval floors at roughly 0.44 — well under any sensible threshold. The run is **informational**, not a pass. No greener than the evidence.

Check	Dimension / tier	State	Value
Faithfulness (generic)	catalog metric	informational	~0.9x
Answer-relevancy (generic)	catalog metric	informational	green
Refund-policy adherence	domain-soundness · judge-rubric	proposed / informational (3/3, lower bound ~0.44)	null

Table 2. Honest scorecard rows. The proposed check carries `value=null` and shows its state; a non-scored metric is never rendered as a fabricated 0.0.

Then the human acts. They read the traces, judge the rubric against their own labeled examples until the judge is calibrated against human ground truth, and only then promote the check to a gate at a threshold they choose. Calibration turns a judge that yields evidence into one that can carry authority; the promotion is a deliberate, host-owned config edit, committed to the repo.

The generic suite passed. The check that mattered was one you had to derive from your own policy — and even then, the evidence, not the estimate, decided when it could gate.

What ships is the frozen, versioned suite. Discovery proposed the check, Core runs it, and the refund-policy gate keeps running in Core whether or not you ever pay Discovery again — because the thing you trust is the scorecard, and you own it.

8 What you own

No engineer will let a non-deterministic agent grade their application on every commit. The objection is correct, and it is the one this paper must answer before it can ask for trust: an agent that reasons freshly each time, and marks its own homework, is not something you gate a release on. So we do not.

The resolution is a split between design time and run time. The non-deterministic reasoning happens *once*, at authoring time, when Discovery deep-reads your code and proposes checks. Its output is not a live grader. It is a frozen, deterministic, versioned Core suite — code plus config — committed to your repository under Apache-2.0. What runs on every commit is that owned Core suite. No agent, and no commercial product, is in the loop at run time.

This matters because it changes what you are trusting. You are not trusting an AI to be right on Tuesday. You are trusting a file you can open, read, and reason about — the same way you trust any other code in your repository. The Evaluation Core that executes it is effect-free: no file, network, clock, or model access. Given the same inputs it returns the same verdict.

The suite is an asset, and it appreciates

Because it is ordinary code beside the application it measures, the suite behaves like ordinary code. It is diffed in pull requests, reviewed by a colleague, reverted when wrong, and blamed line by line. A new check arrives as a readable change, not a silent update from elsewhere.

It also appreciates. Each real production failure, once understood, becomes a check the suite remembers — so the system cannot silently regress into that same failure again. Understanding *why* a hard failure happened — reproducing it, isolating its cause, proving the fix holds — is itself difficult work, and it is the work **EvalGlass Intelligence** exists to do; what it leaves behind is, once more, an ordinary Core check the suite keeps. Over time the suite becomes the system's memory of its own failure modes: not a snapshot of quality, but an accumulating record of every way this application has been shown to break.

If EvalGlass vanished tomorrow, your suite still runs.

Contrast this with metrics rented from a hosted platform. There, the checks live on someone else's servers, computed by someone else's code, under someone else's roadmap. They are gone the day you churn, and gone the day the vendor deprecates them. You cannot diff them, cannot revert them, and cannot read why a number moved. The suite EvalGlass leaves you is yours in the only sense that survives: it is in your repository, and it runs without us.

> **The honest boundary.** Owning the suite is not owning a guarantee. A green run proves only what its promoted checks actually measured — nothing about the failures no one has written a check for yet. The suite is durable evidence and durable memory; it is not a certificate of correctness, and we will not let it read as one. The object you own, and the object you trust, is the scorecard — never the model.

9 Honest limits and design commitments

Project-fitted evaluation costs more than downloading a catalog, and it makes fewer promises. We state those costs and refusals here, in their own section, because this is where authority actually compounds: a method you can trust is one that tells you plainly what it will not do for you.

Derivation requires reading your system. Picking `faithfulness` from a list takes a minute; deriving checks from your code, schemas, prompts and traces takes a deliberate authoring session. This is more up-front work, by design — the price of measuring your failures rather than a generic proxy for them.

Discovery proposes candidates, not a guarantee of completeness. It surfaces *a metric you did not know to write* — not every such metric. We make no claim that a proposed catalog covers all the ways your app can fail; it is a considered starting point for your judgment, and everything it proposes lands proposed and informational.

Proposed metrics need a human to promote them. There is no `approve` verb and no automatic gate: activating one is a deliberate edit you own. A judge-rubric metric inherits the limits of its calibration — until it is calibrated against human ground truth it yields evidence, never authority, and we will not let it gate before then.

A fresh run is `informational`, never a pass. With no active approved gate, the verdict is not green — an absence of judgment, honestly rendered, rather than a manufactured one. A non-scored metric carries `value=null` and shows its state; it is never a fabricated `0.0`.

We refuse to manufacture a trust signal you have not earned. That refusal is the method, not a caveat to it.

The trust object is the scorecard, never the model. EvalGlass does not certify your AI, and it never will — `scorecard.json` records what the evidence supports on the traces you ran, nothing wider. A different distribution tomorrow is a different question, and the honest answer is to measure it again.

> **On the wider family.** Core — the open-source substrate — is available today, and this paper's whole method can be practised in it alone. **Discovery** (which reads a bounded representation of your application and proposes the failure-mode-anchored evaluations it is missing) and **Intelligence** (which diagnoses why a hard failure happened and turns it into a durable evaluation) are separate, paid, pre-alpha / design-partner products, and neither is generally available. We name them here for honesty and show neither producing a customer result; whatever they propose lands as a proposal you review and leaves you an ordinary Core suite, and the product that creates evidence never grants itself the authority to make the consequential decision.

None of these are shortcomings we hope you overlook. They are the design — the shape of a method that stays *no greener than the evidence*, and the most it can offer in a document meant to tell you the truth.

10 How to start

If we have earned your assent to one idea — that the failures worth catching are defined by your application, not by a catalog — then the first step is small, and it is not a sales call.

EvalGlass Core is open source under Apache-2.0, available today. You do not book a demo to begin; you install a plugin in your coding agent and point it at your own repository.

The shortest honest path starts free and local. Install Core in your coding agent — Claude Code today, with Codex as a second runtime — and point it at your repo. Ask it to run the bundled example, so you can read a real `scorecard.json` and see how a verdict is shaped. Then author your first project-fitted check — the one for the failure you already fear — and watch it gate on the evidence rather than the point estimate. When you want the harder finding done for you, the checks you would not have thought to write, that is what **Discovery** proposes; when a failure still slips through and you need to know why, that is **Intelligence**. Both hand back ordinary Core artifacts you keep.

Whether you authored a check or Discovery proposed it, what a first run reports is informational, and nothing gates until you say so: you read the candidates, you decide which — if any — become a gate, and activating one is a deliberate edit you own. Proposal is never approval.

The first scorecard you read will likely be `informational`, not a green tick — that is the point. It reports what it measured, wrapped in a confidence interval, and no more.

That is the whole spirit of this paper: evaluation you can stand behind, because it is derived from your system and never claims more than it measured.

References & further reading

1. E. B. Wilson. "Probable Inference, the Law of Succession, and Statistical Inference." *Journal of the American Statistical Association*, 22(158), 1927. (The Wilson score interval used for proportion scores.)
2. Student [W. S. Gosset]. "The Probable Error of a Mean." *Biometrika*, 6(1), 1908. (The *t*-interval used for mean scores.)
3. Y. Liu et al. "G-Eval: NLG Evaluation using GPT-4 with Better Human Alignment." 2023. (Representative of the LLM-as-judge rubric metrics we credit and build beside.)
4. S. Es et al. "RAGAS: Automated Evaluation of Retrieval Augmented Generation." 2023. (Representative of the reference-free RAG metrics discussed in §2.)
5. EvalGlass documentation — reading a scorecard, statistical honesty, what green does not mean, and how discovery works.
6. EvalGlass Core source — github.com/EvalGlass/evalglass-core (Apache-2.0).

Measurement-science grounding

Evaluation is measurement; the discipline this paper applies is not new. We use these established ideas – we do not claim to have proven validity or reliability for any system.

1. L. J. Cronbach & P. E. Meehl. "Construct Validity in Psychological Tests." *Psychological Bulletin*, 52(4), 281–302, 1955. (A measurement is valid only for the specific construct it was built for – the grounding for project-fitted checks.)
2. S. Messick. "Validity of Psychological Assessment." *American Psychologist*, 50(9), 741–749, 1995. (Names the two threats invoked in §3: construct underrepresentation and construct-irrelevant variance.)
3. F. M. Lord & M. R. Novick. *Statistical Theories of Mental Test Scores.* Addison-Wesley, 1968. (True-score and reliability theory behind the epistemic core: a score without a reliability estimate is not evidence.)
4. I. D. Raji, I. E. Kumar, A. Horowitz & A. D. Selbst. "The Fallacy of AI Functionality." *ACM FAccT*, 2022. (A benchmark pass does not establish that a system works for a specific purpose.)
5. I. D. Raji, E. M. Bender, A. Paullada, E. Denton & A. Hanna. "AI and the Everything in the Whole Wide World Benchmark." *NeurIPS Datasets & Benchmarks*, 2021. (General benchmarks are not general-purpose measures of fitness for use.)
6. A. Karpathy. "Software 2.0." Essay, 2017. (The lineage in §4: weights as the new code; this paper takes the next step, to the specification.)

How to cite. EvalGlass. *Project-Fitted Evaluation: Why trustworthy AI evaluation must be derived from the system under test, not downloaded from a catalog.* Version 1.0, August 2026. https://evalglass.com/whitepaper

```
@techreport{evalglass2026projectfitted,
   title  = {Project-Fitted Evaluation: Why trustworthy AI evaluation must be derived from
the system under test},
  author = {{EvalGlass}},
  year   = {2026},
  type   = {Whitepaper},
  version= {1.0},
  url    = {https://evalglass.com/whitepaper}
}
```