Mozilla.ai

When “Debugging” LLMs Becomes Delusion

Rethinking AI Through Systems Engineering

Raz Besaleli
ML Systems Engineer @ Mozilla.ai

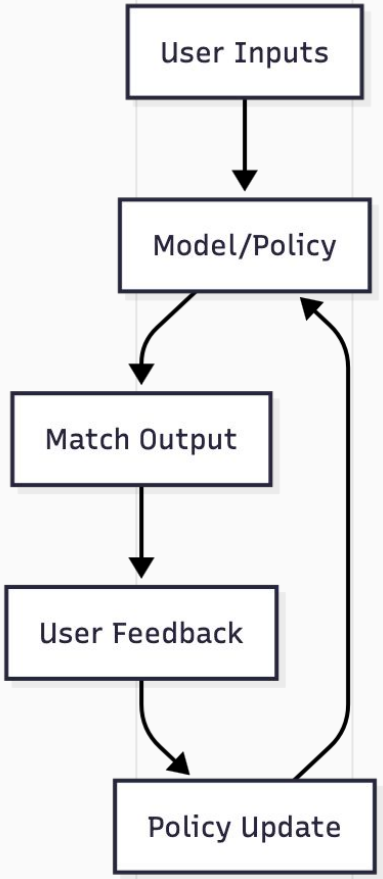
Mozilla.ai

Pretend we're building an AI-powered dating app...

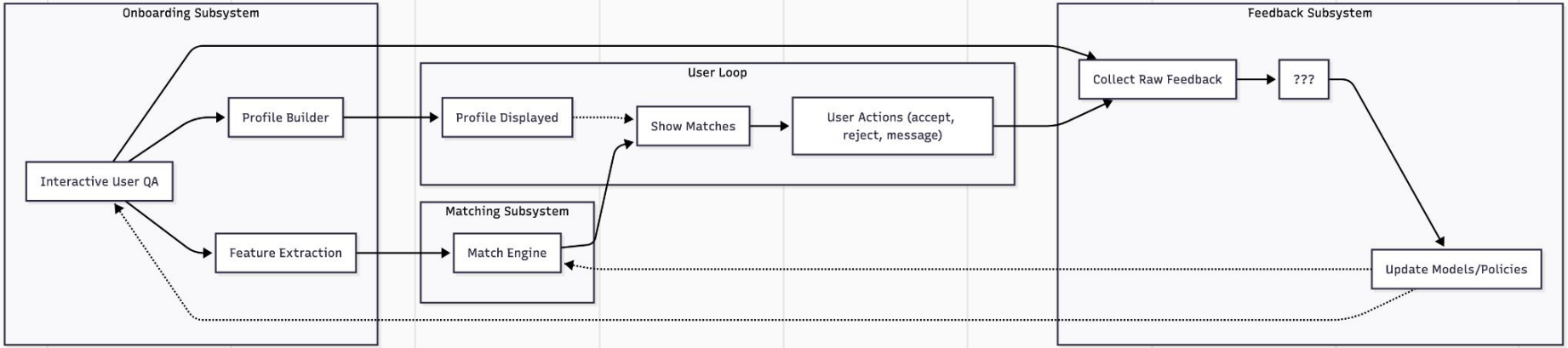
1. GenAI + human-matchmaker logic is used to to build tailored dating profiles
2. Onboarding asks ~50 questions (text/voice) to extract deep user data
3. An algorithm generates matches
4. If both parties agree, they unlock a chat via payment
5. Users provide ongoing feedback on matches to improve the system
6. ??? → Profit!

The Feedback Loop (In Theory)

- **Generalized policy cycle:** engine could be a fine-tuned LLM, a ranking algorithm, or any other matching logic.
- **Model updates can be manual or automatic:** human-driven fine-tuning, active learning, etc.
- **Feedback is the driving signal:** user behavior directly shapes future model behavior *in some way*

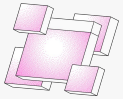


Zooming in a little...



... What could go wrong? 😜

Unstable coupling between core subsystems



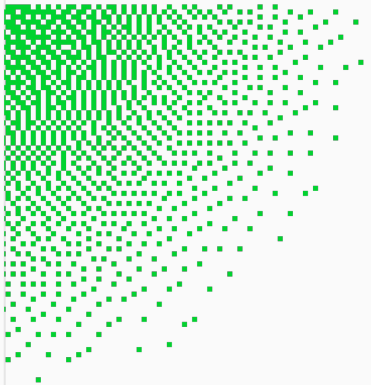
Emergent Homogenization and behavioral collapse

Hidden safety hazards in the loop

Misaligned incentives that may not reflect reality

Unstable Coupling Between Core Subsystems

- Onboarding and matching are tightly coupled through a **dynamically unstable, feedback-amplified, natural-language interface**.
- The impact of onboarding's interactive QA on downstream matching is **hard to measure or isolate**
- **When users change how they answer** (natural variation, gaming, shortcuts, over-sharing, under-sharing), **the matching engine shifts too** — and you can't disentangle why or where it's happening

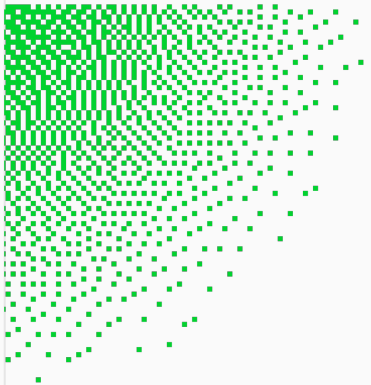


NOTE: These are challenges, not non-starters—but you have to treat them as engineering constraints, not afterthoughts.

Emergent Homogenization and Behavioral Collapse

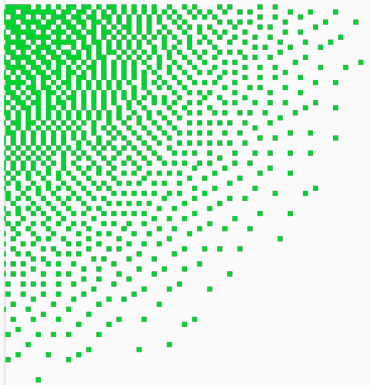
- Eventually, the matching algorithm learns that 'success' **means matching people with conventionally attractive profiles** and starts to form an **emergent monolith of attraction**
- The system starts to eat itself:
  - Everyone starts to get shown the same 10 faces
  - Users reshape their profiles to fit the “successful” archetype, undermining the system’s original behavioral assumptions.
  - People start to burn out on algorithmic “sameness”
  - Long-tail users start to leave: ‘this app isn’t built for people like me’

Hinge’s “most compatible,” Bumble’s “spotlight balancing,” and Tinder’s moves away from pure ELO-style scoring all came **years** after the industry realized the system was collapsing.



Hidden Safety Hazards

- Training data reflects user behavior—for better or for worse.
- Discriminatory outcomes (e.g. certain demographics systematically ranked lower) can breach anti-discrimination law even if the bias is “emergent”
- Users don’t consent to being part of “algorithmic experiments,” yet most apps quietly A/B test attraction models in real time
- Under the **EU AI Act,** a dating app’s recommendation system could be classified as “high-risk” if it significantly affects people’s social lives or opportunities (which it does)

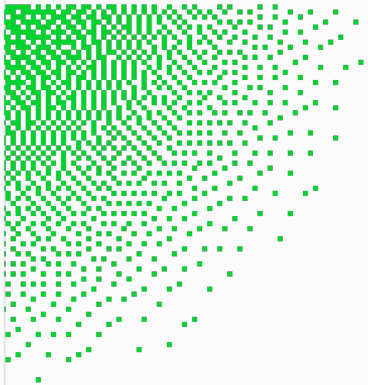


Misaligned Incentives

What They Say: “If we make people pay per match, it will weed out those with questionable incentives”

What Actually Happens:

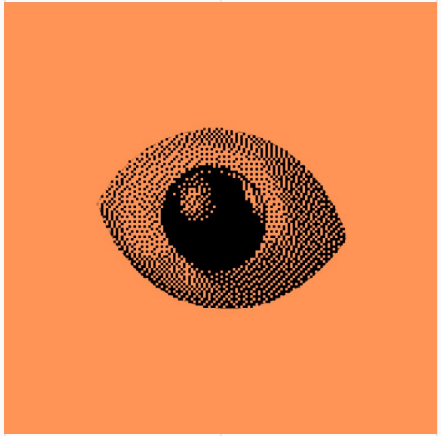
- The system monetizes *validation*, not compatibility
- Turns loneliness into a revenue source, not a problem to solve
- Rewards the system the most when users never quite succeed
- Attracts high-intent spenders, not high-intent partners
- Creates a perverse loop: **pay → visibility → more matches → more pay → skewed ecosystem**



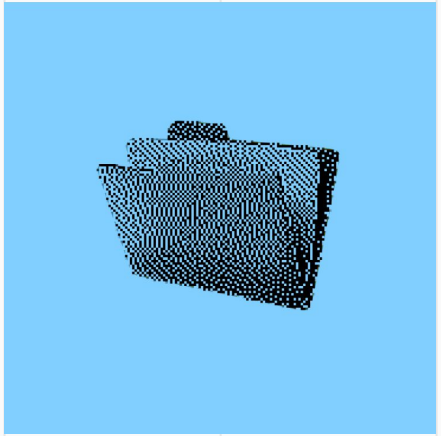
Mozilla.ai

The missing role

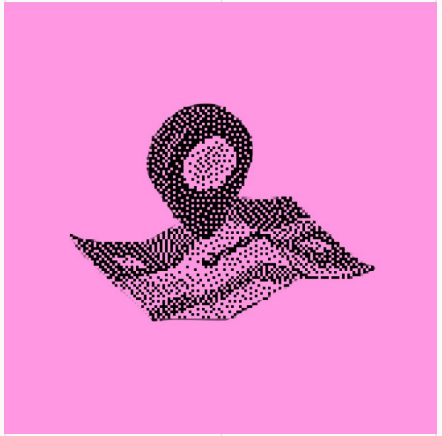
Who...



... owns risk across the whole system?

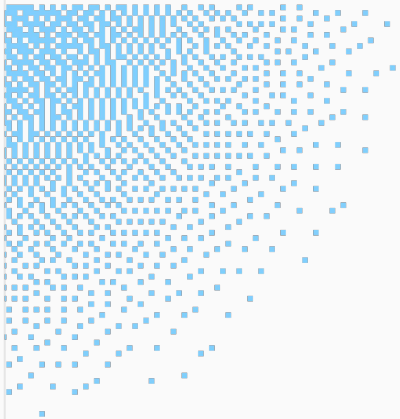


... tracks how these feedback loops interact?



... is responsible for overall system behavior, not just its parts?

Introducing the ML Systems Engineer



✨ The Experiment ✨

We've been running an experiment at Mozilla.ai where we hired someone to...

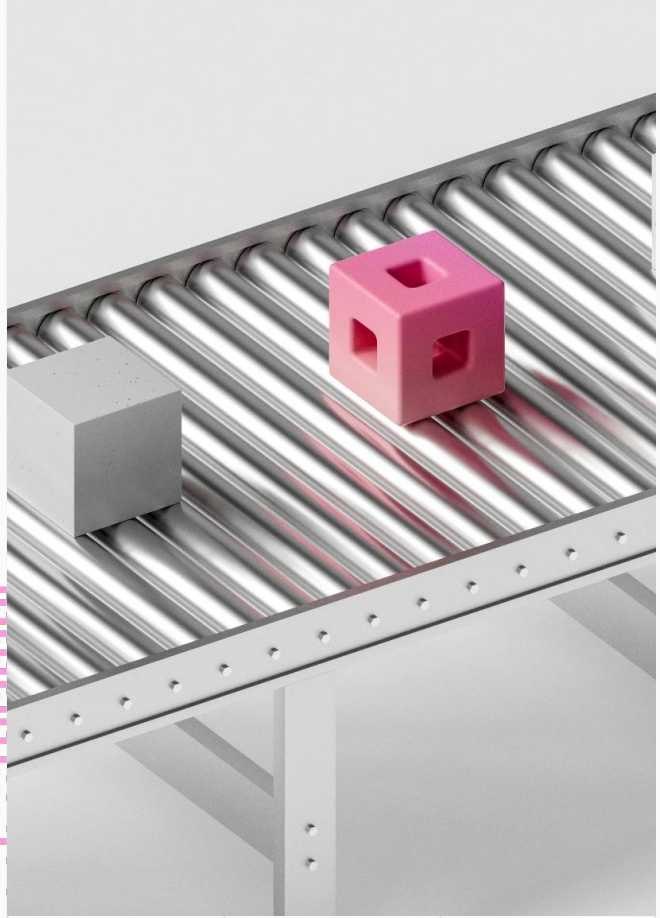
- Explicitly own **system-level integrity**;
- Prototype cross-functional processes for spotting and mitigating risks;
- Build institutional memory of system behavior beyond model metrics.

THE LAB RAT :)

The one job: de-risk.

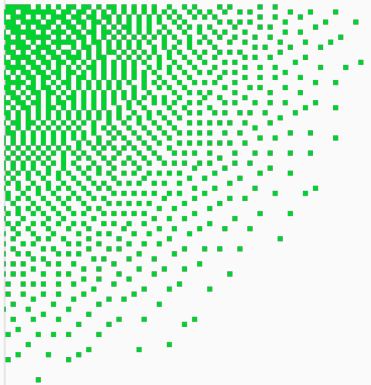
Systems Engineering: The Lore

- **Born in aero and defense (1940s-1960s)**
  Emerged when projects like missile guidance, radar networks, and Apollo made it impossible to treat components in isolation — failures were systemic, not local.
- **Created to manage complexity across domains**
  Integrated hardware, software, communications, human operators, logistics, and mission constraints into a single, coherent system — something no individual team owned.
- **Focused on preventing failure**
  Systems engineers were responsible for interfaces, feedback loops, failure propagation, and lifecycle behavior — not the components themselves, but the space between them.



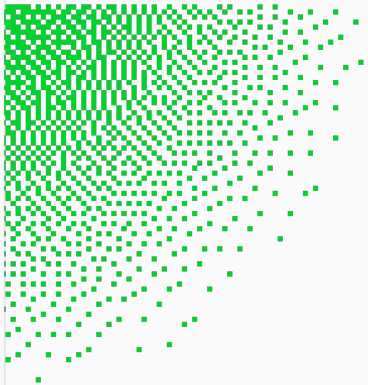
Why Software Forgot Systems Engineering

- **Software ate the world faster than systems engineering could follow**
  Startups optimized for velocity, not lifecycle traceability or cross-domain coherence.
- **Modern roles evolved to focus on uptime, not system behavior**
  SRE, DevOps, and MLOps are operational disciplines — focused on keeping things running, not understanding how failures propagate.
- **“Move fast and break things” replaced “model the system”**
  Software culture treated failure as cheap and reversible, assuming deterministic components instead of adaptive, emergent ones.



But, AI is different from traditional software.

- **AI systems aren’t deterministic — they’re adaptive and coupled**
  Models learn from behavior, behavior learns from models, and the whole system evolves unstably over time.
- **Failure is emergent, not local**
  Bugs appear as feedback loops, drift, incentive misalignment, and sociotechnical dynamics — not stack traces.
- **Modern compliance needs (and product reliability) mandate system traceability**
  If you can’t show how decisions were made, you can’t show they were allowed
- **No one owns the interfaces, assumptions, or propagation paths**
  Exactly the space that systems engineers were created to manage.

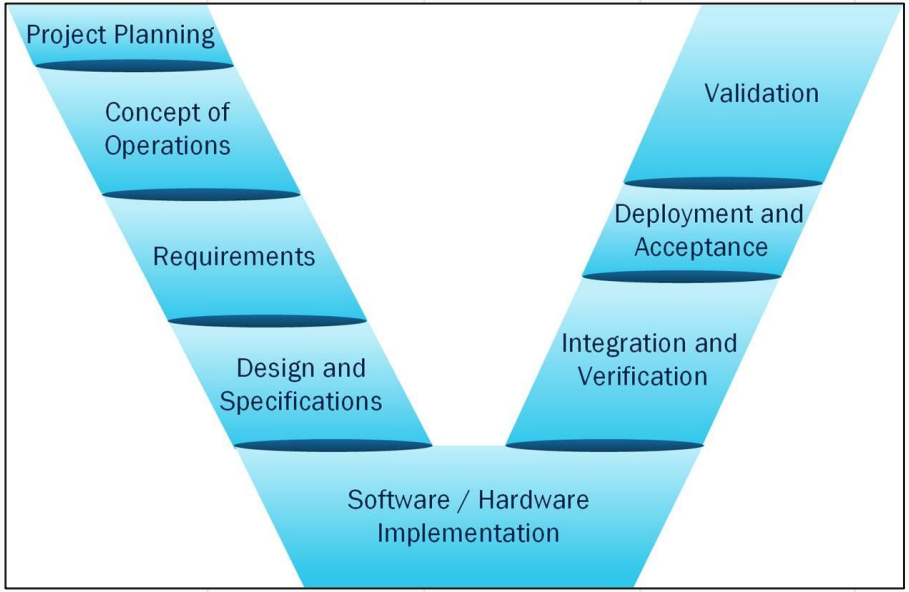


Ok, but what is SE, actually?

SE comes in many flavors, but they all orbit the same idea: **think about what you're building for, like, five minutes before you actually build it.**

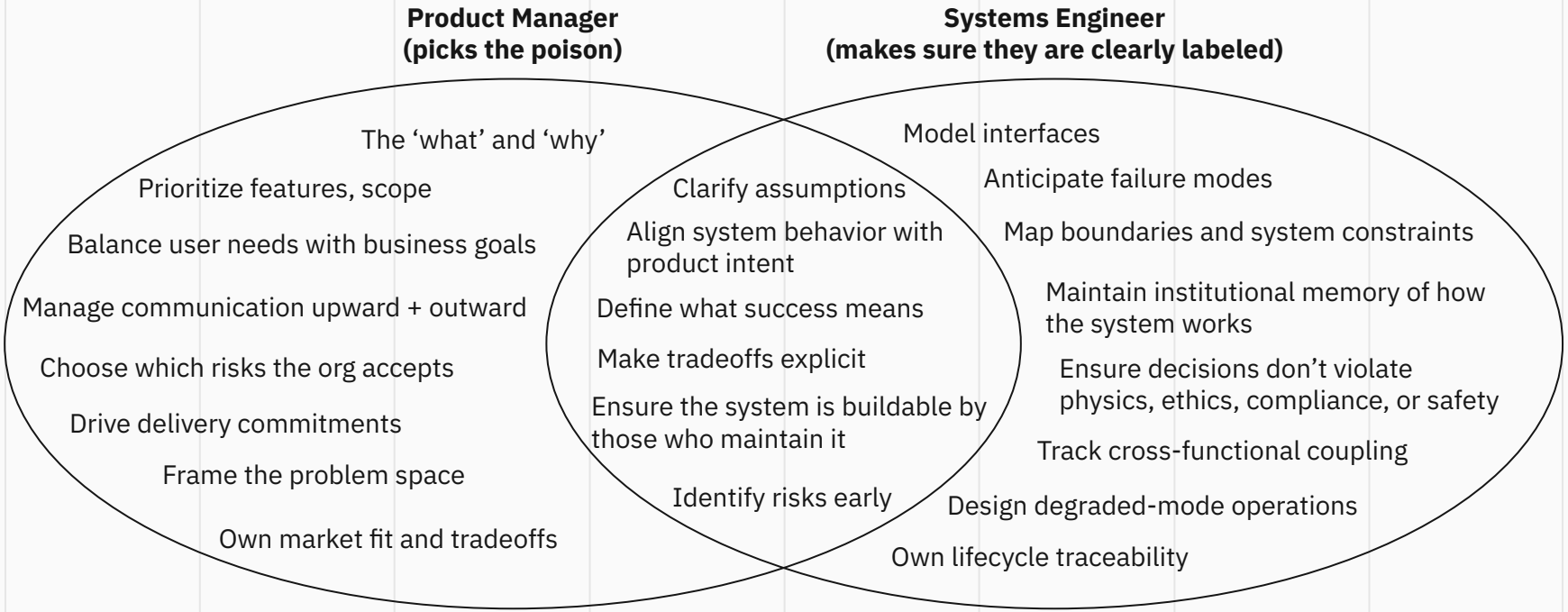
Variations include:

- **Document-based (DBSE)**
  Requirements, interfaces, and risks live in formal docs
- **Model-based (MBSE)**
  Everything becomes a diagram, for better or worse
- **Human Systems Engineering**
  Treats people as integral subsystems, not afterthoughts
- **Agile Systems Engineering**
  The whole V recreated each sprint, but lightweight.
- ... the list goes on

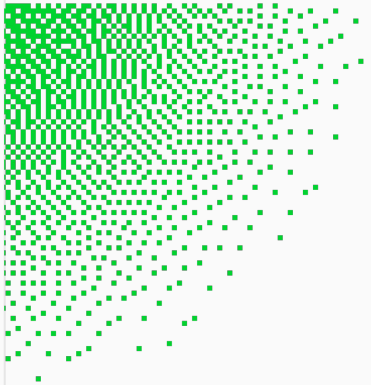


Mozilla.ai

Systems Engineers vs Product Managers



What Works & What Doesn’t



Project Planning & CONOPS

Traditional SE

Project Planning

- Define mission goals upfront
- Freeze high-level requirements
- Establish constraints early
- Plan the entire lifecycle end-to-end

CONOPS

- Formalized use-cases and scenarios
- Defined actors, responsibility flows
- Stable operational modes
- Assume predictable human behavior

Machine Learning SE

Project Planning

- Define system boundary... *provisionally*
- Capture assumptions, not promises
- Identify dangerous couplings *early*
- Design for drift and iteration

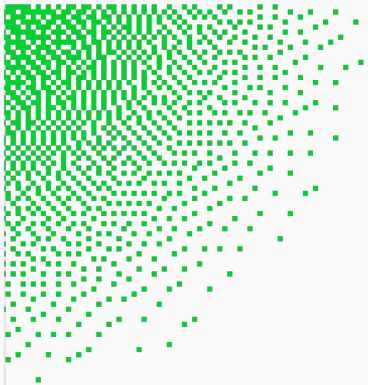
CONOPS

- Intended use, not frozen workflows
- Humans as variable, unreliable sensors
- Multiple operational modes (*especially* bad ones)
- Plan for emergent behavior

Example — Project Planning & CONOPS

“We’d like to build semantic search using RAG across all internal documents so employees can find information faster.”

What is wrong here?



Example — Project Planning & CONOPS

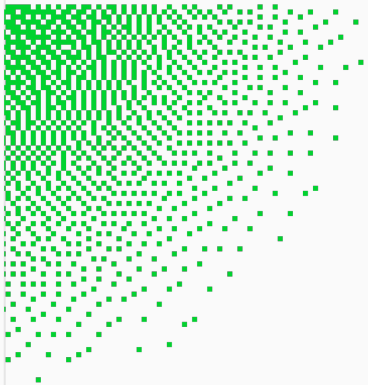
“We’d like to build semantic search using RAG across all internal documents so employees can find information faster.”

What is wrong here?

Why are we jumping to RAG when we don’t even know…

- Whether we can even deploy an LLM or use a third-party API
- Whether hallucinations are even acceptable for this domain
- If we’re legally allowed to paraphrase documents
- How fast our knowledge base changes
- If chunking the data is viable
- Whether retrieval is even the bottleneck

It may be the case that BM25 retrieval system + a small, span-based QA model may be better.



Requirements

Traditional SE

- **Requirements Traceability**
  Keep link between intent -> design -> test -> operations
- **Clear Functional Requirements**
  Stating what the system must do—not how
- **Performance Requirements**
  Latency, throughput, accuracy thresholds
- **Constraints & Assumptions**
  Document physical, organizational, technical limits
- **Verification Planning**
  Defining measures of performance (MoPs) and measures of effectiveness (MoEs)

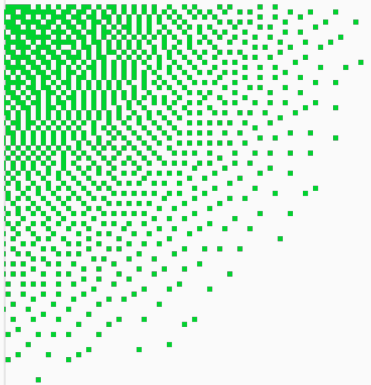
Machine Learning SE

- **Degraded Mode Requirements**
  The system *will misbehave frequently*. What should happen when it does?
- **Document Assumptions**
  You will have lots.
- **Anti-Magic Requirements**
  Every end-to-end functionality of the system should be *observable* and *falsifiable*
- **Human-in-the-loop Requirements**
  How and when do people intervene, override, or course-correct?
- **Data Sourcing, Drift, and Freshness**
  How do we keep the system up-to-date?

Example — Requirements

“The system shall generate SQL queries from natural language with 95% accuracy.”

What is wrong here?



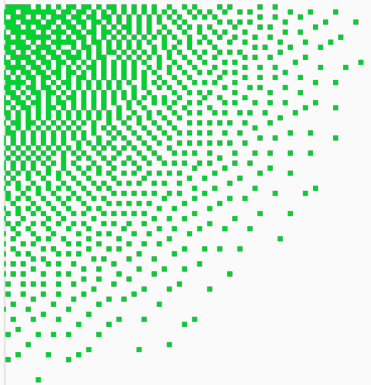
Example — Requirements

“The system shall generate SQL queries from natural language with 95% accuracy.”

What is wrong here?

Accuracy of *what*, exactly?

- SQL correctness isn’t binary: syntax, semantics, joins, cardinality, permissions, cost.
- A query can be “correct” syntactically and still return nonsense or leak data.
- “Accuracy” means something different to users, analysts, DBAs, and ML folks — which one is this requirement talking about?
- And are we assuming it will generate some SQL 100% of the time? What counts as “accurate” when the model refuses?



Design & Specification

Traditional SE

Goal: Build a system whose components interact cleanly and predictably

Design Assumptions: Interface is stable, behavior is deterministic, and components behave correctly unless broken

Design Strategies:

- Clear subsystem boundaries
- Frozen interfaces
- Trade studies
- No hidden side effects
- Integration is primarily a *coordination* problem

Machine Learning SE

Goal: Build a system whose failures are *observable* and *contained*

Design Assumptions: The model is unpredictable, “correct” behavior is ill-defined, users will behave adversarially

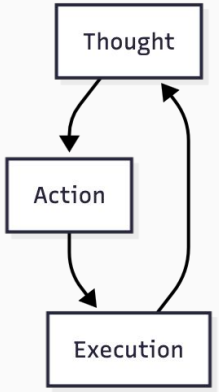
Design Strategies:

- Composability over cleverness
- Guardrail-first design
- Instrumentation, not intuition
- Containment boundaries

Example — Design & Specification

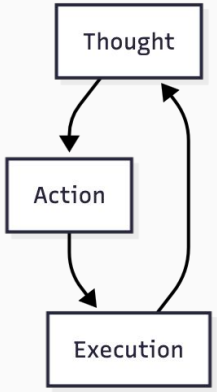
“We’ll let the LLM decide when to use tools based on ReAct. It can call whatever it needs.”

What is wrong here?



Example — Design & Specification

“We’ll let the LLM decide when to use tools based on ReAct. It can call whatever it needs.”



What is wrong here?

- **Ungoverned loops hide failure**
  If the agent can invent steps on the fly, you can’t tell where it went off the rails — only that the output is wrong. An explicitly defined finite state machine forces failure to happen in public.
- **Unconstrained behavior = bad**
  You shouldn’t assume that an agent can select the correct tool to call from an unconstrained set at every step.
- **Safety demands a closed action space**
  If the model can transition to any action at any time, you can’t enforce guardrails.

Verification & Validation

Traditional SE

- **Verification = “Did we build it right?”**
  Tests map cleanly to requirements. Behavior is deterministic.
  Measured with **MoPs**
- **Validation = “Did we build the right thing?”**
  Traceable to mission goals. Human acceptance is stable.
  Measured with **MoEs**
- **Evaluation is finite.**
  Test cases are enumerable and exhaustive. Pass/fail is crisp.

Machine Learning SE

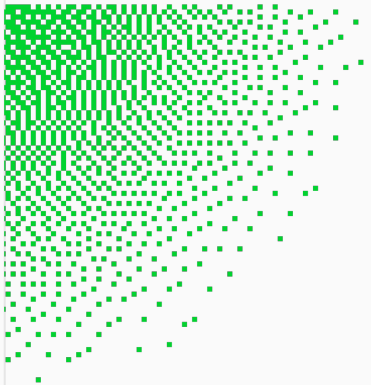
- **Verification requires *coverage,* not cases.**
  Inputs are unbounded. Behavior is probabilistic. You test distributions, not conditions.
- **Validation never finishes.**
  User behavior shifts. Data drifts. Context changes. You don’t validate once — you validate *forever*.
- **Emergent failures demand ongoing evals.**
  Models develop new behaviors post-deployment. Silent degradation is the norm unless you explicitly measure for it.
- **Evaluation must be designed for adversarial behavior**
  Users will misuse and abuse the system in ways you could not even imagine.

Mozilla.ai

Example — Verification & Validation

“We’ll have an LLM evaluate model outputs for correctness.”

What is wrong here?

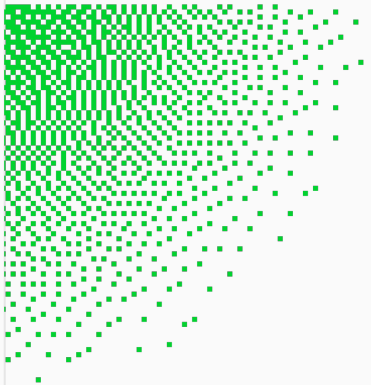


Example — Verification & Validation

“We’ll have an LLM evaluate model outputs for correctness.”

What is wrong here?

- **You are adding more failure modes to the system.**
  If the judge and the model share failure modes, biases, or blind spots (they will), the judge just rubber-stamps errors.
- **LLM judges hallucinate confidence in a way that is dangerous.**
  You are effectively measuring vibes.
- **“Correctness” is not well-defined.**
  An LLM-judge’s scoring depends on phrasing, temperature, token entropy, the list goes on. These don’t yield metrics. These yield weather patterns.
- **Distribution mismatch is lethal.**
  Your judge is validated on curated benchmarks, but deployed on messy, real-world data.

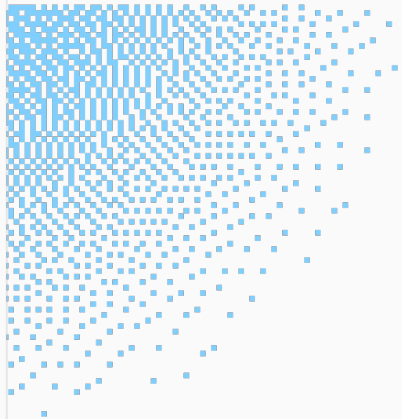


Would you trust an LLM to judge your compatibility with another human?

Great! Regulators wouldn’t, either.

Mozilla.ai

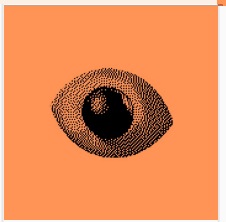
Risk



The Cassandra Complex

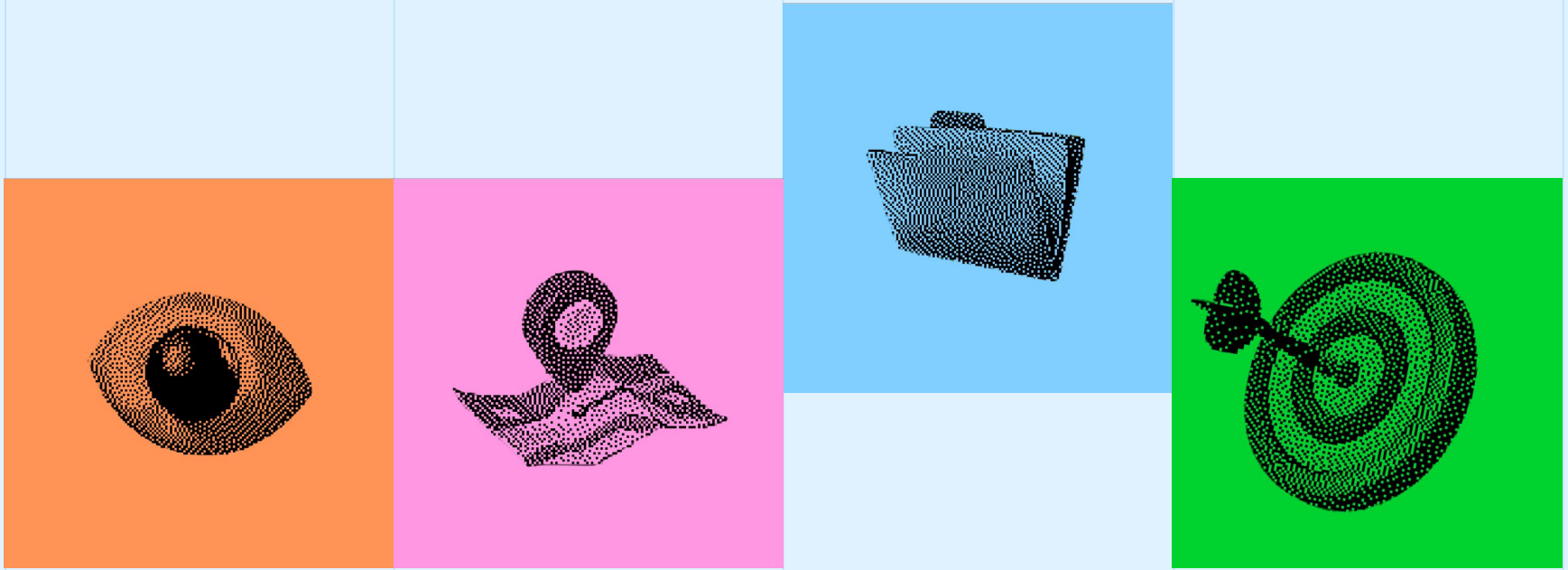
The curse of seeing a failure coming long before anyone else — and not being believed until it’s too late.

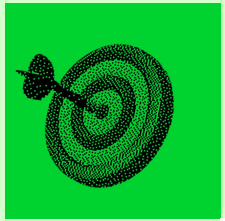
It’s less “prophecy,” more “unwelcome accuracy delivered early.”



Mozilla.ai

Have you ever felt like a Cassandra at an AI company?

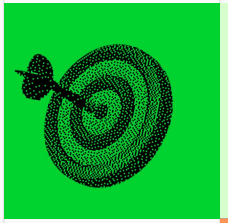




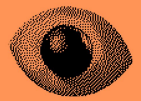
AI companies like to dodge risk.

Mozilla.ai

Having a Systems Engineer... Helps.



Gives structural legitimacy to risk—a person to *own* it



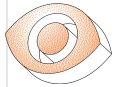
"I have a bad feeling" → "Here's the failure mode, likelihood, and mitigation options"

Adds new dimension to institutional memory

Mozilla.ai

Systems Engineers & Risk

(alt. What they don’t teach you in SE school)



Cassandra duty

Listening to the people who see the cracks before anyone else—especially when no one else will

Jester’s privilege (...or curse)

Saying the quiet part out loud that no one else wants to hear

Risk without panic

Communicating danger without triggering shutdown or denial

Emotional buffer

Holding the anxiety so others can keep building

The Risk Registry

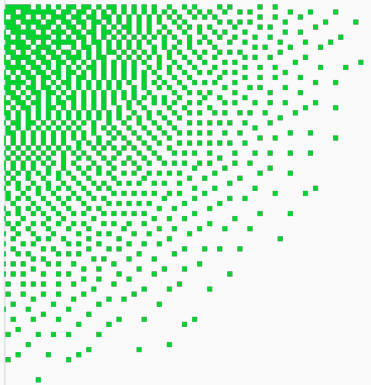
What it is: A living list of everything that could break the system, how likely it is, and what we're doing about it.

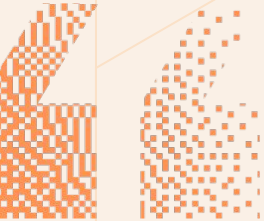
The rules:

- **One owner** keeps the map coherent. Everyone else contributes where they see landmines.
- **Everything** goes in the risk registry. No matter how large or small.
- **Periodically reviewed** at a cadence that works for your team.

Why it works:

- Keeps risk *visible* instead of ambient
- Gives everyone a space to voice concerns and feel heard
- Prevents rediscovering the same failure twice
- Weirdly... speeds up shipping





Stop asking ‘is this going to happen?’

Start asking ‘when it does, what’s our plan?’

Mozilla.ai

Interactive Session

Mozilla.ai

Check out our GitHub!

besaleli@mozilla.ai

@besssaleli on X