Adversarial agents

Adversarial agents are specialized AI systems designed to challenge, attack, or critique other models in order to expose hidden flaws, prevent silent errors, and improve reliability. Why this needs to become a discipline, what it inherits from security and software engineering, and what changes when the systems under test can act on their own.

Matthew J. Harmon

2026-09-15

Working draft — v0.2

The failure that matters is the quiet one

A model that refuses, crashes, or produces obvious garbage is not the hard problem. Those failures are loud; someone notices and they get fixed.

The dangerous failure looks like success: a confident wrong answer, a tool call that does something plausible instead of what was intended, a reasoning chain that skips the step that would have exposed its own mistake, a safety boundary that holds against the obvious probe and fails on the fourth variation.

Anyone who has worked incident response knows the pattern. The breach that costs you is the one that sits in the environment because nothing is looking for the right signal. Detection is not added after the system is built; it is a posture designed in from the start.

Adversarial agents apply that posture to AI. Their job is to make silent failures observable before they reach the world.

Why static tests are not enough

Benchmarks are necessary but fixed. Once a test set exists it becomes a target: models are trained toward it and learn to pass it without necessarily acquiring the property it was meant to measure. Over time the test measures familiarity as much as competence.

An adversary has an objective that conflicts with the target's. It searches, changes tactics when one approach fails, and pushes harder on inputs that came close. Its output is not a score but a concrete, reproducible case where the target behaved incorrectly.

Security has understood this for decades. A vulnerability scanner checks known conditions. A penetration tester looks for a way through. The same distinction applies to AI systems.

Four roles

Each role needs different capabilities, placement, and measures of success.

- **The attacker** generates inputs intended to cause failure: jailbreaks, prompt injection through retrieved or tool-supplied content, coercing an agent into misusing capabilities it legitimately has. *Success: a reproducible exploit.*
- **The critic** examines outputs for unsupported claims, dropped constraints, arithmetic that does not check, code that compiles but behaves incorrectly, conclusions that outrun their evidence. *Success: a mistake caught before it escapes review.*
- **The challenger** attacks the conclusion rather than the interface. Given an answer or plan, it builds the strongest case against it and forces the target to defend its reasoning. *Success: a retracted, revised, or battle-tested conclusion.*
- **The monitor** observes an agentic run in progress — tool calls, state changes, intermediate decisions, divergence from the stated task — and does not wait for the final answer. *Success: a bad action stopped before it becomes irreversible.*

A mature program needs all four. Current work is concentrated on attack generation and, increasingly, output critique. Continuous challenge and runtime monitoring remain much less developed.

Lineage

The ideas are not new; adversarial agents bring together traditions previously applied to different kinds of systems.

Tradition	What it contributes
Generative adversarial networks	Opposition as part of the learning structure — one system improves because another is rewarded for catching its failures
Debate and critique	Structured disagreement between systems as evidence for a judge
Fuzzing and mutation testing	The attacker role applied to software — inputs the author never anticipated
Chaos engineering	Deliberate failure injection so resilience is tested before real conditions test it
Red, purple, and tabletop teams	The human precedent — independent parties contesting assumptions and probing defenses

Adversarial agents extend these to systems that reason, use tools, maintain state, and act at machine speed.

Design principles

Treating adversarial agents as operational systems rather than evaluation tricks implies the following.

Independence. An adversary that shares too much with the target shares its blind spots. Different model families, prompts, retrieval paths, tools, and context create useful separation. Local, controllable models help here because their context, tool access, and logging can be constrained and inspected.

Conflicting objectives. If both systems optimize for "the best answer," the result is collaboration, not pressure. The attacker is rewarded for finding failure, the critic for unsupported claims, the challenger for broken conclusions, the monitor for early detection of divergence. The disagreement is intentional.

Record the contact point. The interaction between target and adversary is where the evidence is produced. Capture inputs, outputs, tool calls, state, and context well enough to reconstruct what happened. A finding that cannot be reproduced is hard to distinguish from an anecdote.

Measure more than successful attacks. An adversary that flags everything, or only finds known failures, is not useful. Track reproducibility, false-positive rate, novelty, severity, time to detection, and whether findings lead to durable controls rather than narrow patches.

Control cost. Adversarial systems consume inference, latency, engineering effort, and attention. Deploy them where the expected cost of failure justifies the overhead: attackers during development, critics around important outputs, monitors around high-impact actions.

Adversaries fail too. The adversarial layer is another model-driven system. It can miss failures, invent them, become predictable, or drift toward the target's assumptions. It needs evaluation, rotation, and challenge of its own, or it becomes the next source of silent failure.

From evaluation to operations

For a chat model, a wrong output may be a wrong sentence. For an agent with tools, it may be a wrong action: a file deleted, a message sent, a transaction submitted, a credential disclosed, a configuration changed. The cost of silent failure grows with the agent's reach.

That moves adversarial work out of pre-release evaluation and into the operating architecture. A high-impact action may need a second system to challenge its justification before execution. A long-running task may need an independent monitor. A generated report may need a critic before anyone acts on it. Repeated findings should feed back into policy, detection, tooling, and design.

This is closer to a security operations model than a test suite. The point is not to prove once that the system is safe; it is to keep looking for evidence that it is not.

That is the discipline: systems designed to oppose one another in controlled ways, with enough instrumentation to understand what happens at the point of contact.