Innocito

THE COMPLETE PLAYBOOK FOR TESTING AI AGENT SYSTEMS

- PROCEDURES
- CHECKLISTS
- TOOL GUIDES
- DECISION FRAMEWORKS

TABLE OF CONTENTS

01	Why AI Systems Need a Different Testing Approach	03
02	The Innocito AI Testing Framework	05
03	Phase 1: Agent Unit & Prompt Testing	06
04	Phase 2: Multi-Agent Workflow & Integration Testing	08
05	Phase 3: Context & Memory Validation	10
06	Phase 4: Adversarial & Scenario Fuzzing	11
07	Phase 5: Resilience & Chaos Testing	12
08	Phase 6: Performance, Load & Soak Testing	13
09	Phase 7: Guardrail & Safety Validation	15
10	Decision Trees: Choosing the Right Test	16
11	Checklists & Templates	18
12	Tool Quick-Start Guides	21
13	Engagement Model: Working with Innocito	27

01

Why AI Systems Need a Different Testing Approach

If you've spent years testing traditional software, AI agents will humble you quickly. The assumptions that make classic QA work (same input, same output, assert-equals) simply don't hold here. The same prompt can return a different answer on every run, agents decide for themselves which tools to call, and multiagent workflows produce emergent behavior that no single unit test can predict.

The Five Testing Gaps In AI Systems

THE INNOCITO PERSPECTIVE

None of these gaps closes on its own, and none of them yields to traditional QA. What works is a purpose-built quality engineering practice: AI-native evaluation techniques paired with enterprisegrade process rigor. That's exactly what the Innocito AI Testing Framework is built to give you.

02

The Innocito AI Testing Framework

We organize AI testing into seven progressive phases, each building on the one before it. The logic is simple: catch every defect at the earliest, and cheapest, stage where it can be caught.

PHASE	WHAT IT VALIDATES	KEY TOOLS
1. Agent Unit & Promp	Individual agent correctness, prompt stability, schema compliance	Pytest, DeepEval, LangSmith
2. Workflow & Integration	Multi-agent handoffs, orchestration, planner-executor loops	LangSmith, Playwright, custom harness
3. Context & Memory	RAG retrieval, context window, memory consistency, long conversations	Ragas, TruLens, MTEB
4. Adversarial & Fuzzing	Prompt injection, malformed inputs, edge cases, contradictions	Garak, PyRIT, Promptfoo
5. Resilience & Chaos	Failure recovery, retries, circuit breakers, degraded operation	Toxiproxy, Chaos Toolkit
6. Performance & Load	Latency, throughput, scalability, soak stability	Locust, ks, OpenTelemetry
7. Guardrails & Safety	PII, toxicity, hallucination, policy compliance, output scope	Presidio, Garak, Perspective API

The sections that follow walk through each phase in detail: step-by-step procedures, concrete examples, tool configurations, and checklists you can put to work the same day.

On the numbers ahead:

Thresholds like a 95% task success rate, a sub-2% hallucination rate, and P95 latency under 8 seconds aren't universal constants. They're the starting points we calibrate to across AI testing engagements, defensible defaults, not guesses. Tune them to your own domain's risk tolerance; treat them as a baseline to argue with, not a certification to pass.

03

Phase 1: Agent Unit & Prompt Testing

Validate individual agents in isolation before they participate in any workflow

Step-by-Step Procedure

1

Catalogue All Agents and Their Contracts

- Document each agent's input schema, output schema, system prompt, and tool access list. Maintain this catalogue in a shared wiki or API spec (OpenAPI / JSON Schema).

2

Build Evaluation Datasets

- Create a minimum of 50 labeled input/expected-output pairs per agent. Include: happy path (60%), edge cases (20%), adversarial inputs (10%), boundary conditions (10%). Tool: LangSmith Datasets or Argilla for collaborative dataset curation.

3

Implement Automated Evaluators

- For each agent, define evaluation criteria: structured output uses exact match or JSON schema validation; free text uses semantic similarity (cosine ≥ 0.85) plus ROUGE-L ≥ 0.70; tool selection asserts the correct tool and valid parameters for each test case. Tools: DeepEval (metrics library), Pytest (runner), LangSmith (tracing and evaluation).

4

Run Prompt Regression Suite

- Execute the evaluation dataset against the current prompt version. Store results with the prompt version hash. Any prompt change triggers re-evaluation; alert if accuracy drops more than 2% versus baseline. CI integration: add as a GitHub Actions / Jenkins step that blocks merge on failure.

5

Multi-Run Statistical Validation

- Because LLM outputs are non-deterministic, run each test case 3 to 5 times. Use majority vote or mean score to determine pass/fail. Flag cases with high variance (more than 20% spread) for prompt refinement.

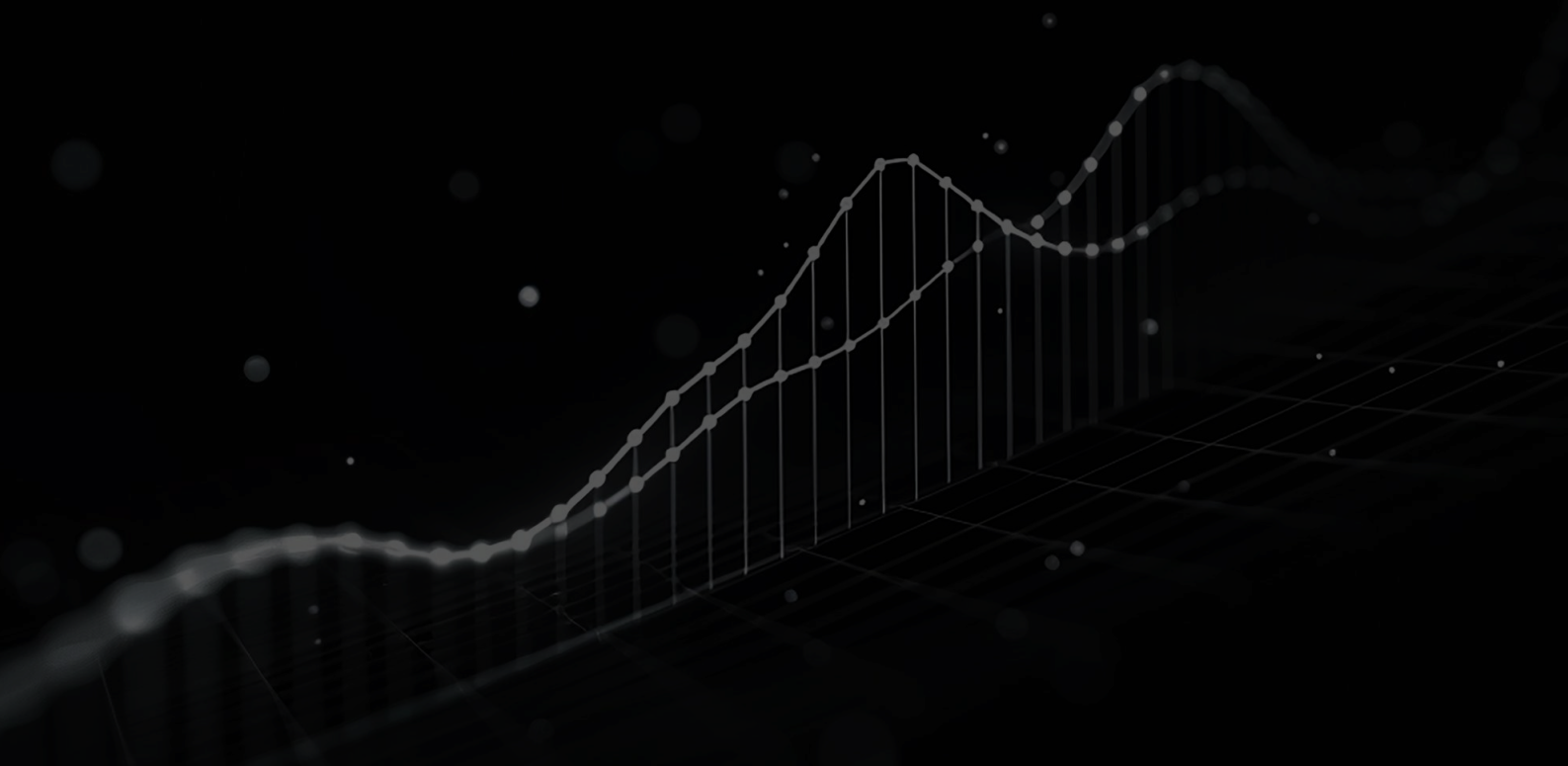
Example: Summarization Agent Unit Test

Agent: SummarizerV2 | Prompt version: v2.3.1 | Dataset: 80 labeled articles

INPUT	A 500-word tech article about quantum computing.
EXPECTED	Summary under 100 words, mentions 'qubit', 'entanglement', 'error correction'. JSON:{summary: string, key_terms: string[]}
EVALUATORS	(1) JSON schema valid ✓ (2) Word count ≤ 100 ✓ (3) Key terms overlap ≥ 80% ✓ (4) ROUGE-L vs. reference ≥ 0.72 ✓
RESULT	4/4 evaluators pass across 5 runs. Variance: 3.2%. Status: PASS

IN THE FIELD

What we see in practice: teams build the happy-path 60% of the evaluation dataset first, ship the agent once it's green, and never circle back to the adversarial and boundary slices. Those are exactly the cases that don't show up in a demo, and do show up in production a few months later.



04

Phase 2: Multi-Agent Workflow & Integration Testing

Validate agent collaboration, handoffs, and end-to-end pipeline correctness

Step-by-Step Procedure

1

Map All Workflow Topologies

- Document every multi-agent pipeline: which agents participate, their order, data contracts between them, and branching logic.
- Deliverable: workflow topology diagram (Mermaid or draw.io) for each pipeline.

2

Define Handoff Contracts

- For each agent-to-agent handoff, define the output schema of the upstream agent, the input schema of the downstream agent, required fields, and transformation logic.
- Test: a contract violation test, deliberately send malformed handoff data and verify graceful error handling.

3

Trace-Based Assertions

- Enable distributed tracing (OpenTelemetry) across the full pipeline. Write assertions against trace data: agent execution order matches the expected topology, each agent received its expected input from upstream output, and no agent was skipped or executed out of order.
- Tools: LangSmith Traces, Jaeger, OpenTelemetry SDK.

4

End-to-End Macro Validation

- Evaluate the final workflow output against ground truth. This validates the complete chain, not individual agents.
- Metrics: task success rate ≥ 95%, cross-agent consistency, goal alignment, information fidelity

Example: Research → Writing → Verification Pipeline

User request: “Produce a 500-word competitive analysis of electric vehicle battery manufacturers, citing 3 sources.”

STEP 1 ASSERT	Research Agent returns at least 3 source objects with URL, title, and relevance score.
STEP 2 ASSERT	Writing Agent output is 450–550 words and references only sources from Step 1.
STEP 3 ASSERT	Verification Agent confirms all claims are supported by cited sources. Hallucination rate: 0%.
MACRO ASSERT	Final doc meets word count, uses exactly 3 sources, tone classifier scores ‘professional’ ≥ 90%

IN THE FIELD

What we see in practice: the handoff contract test is the first thing cut under deadline pressure, and it's almost always the root cause when a pipeline silently returns garbage instead of failing loudly. Writing it up front costs a day. Debugging its absence in production costs a lot more.

05

Phase 3: Context & Memory Validation

Ensure agents retrieve the right information and retain context across interactions

Step-by-Step Procedure

1

Vector DB Retrieval Accuracy Benchmark

- Seed the vector store with a controlled dataset (at least 500 documents). Create 100 query/ground-truth pairs. Metrics: Precision@5 ≥ 0.90, Recall@10 ≥ 0.85, MRR ≥ 0.80.
- Tools: Ragas (evaluation), Pinecone / Weaviate / ChromaDB test harness.

2

Context Window Boundary Testing

- Create prompts that approach 80%, 90%, 95%, and 100% of the model's token limit. Assert that critical information placed at different positions (start, middle, end) is correctly used.
- Tool: custom Pytest parametrized tests with tiktoken for token counting.

3

Memory Staleness Detection

- Update a document in the knowledge base. Query the agent within 1 minute, 5 minutes, and 30 minutes. Assert the agent uses the updated version, not the stale cache. Staleness rate target: under 2%.

4

Long Conversation Regression

- Simulate 20+ turn conversations. At turn 15 or later, reference information from turn 2 to 3. Assert the agent correctly recalls and uses early conversation context. Task success rate: ≥ 95%.

IN THE FIELD

What we see in practice: memory staleness is the gap engineering leaders most underestimate. It doesn't fail a demo. It shows up weeks after launch, when someone asks about a policy that changed and the agent confidently answers with the old one.

06

Phase 4 : Adversarial & Scenario Fuzzing

Probe the system with unexpected, malformed, and malicious inputs to find hidden failure modes

Step-by-Step Procedure

1

Run Automated Vulnerability Scanner

- Tool: Garak. Configure with your model endpoint. Run all default probes: prompt injection (direct, indirect, encoded), jailbreak attempts (DAN, role-play, encoding bypass), data leakage probes (system prompt extraction, training data extraction).
- **Pass criterion**: zero successful injections or jailbreaks.

2

Multi-Turn Red Team Testing

- Tool: PyRIT (Microsoft). Run automated multi-turn attack scenarios. PyRIT simulates an adversary that adapts its strategy across conversation turns; configure it with your model and safety policies. **Duration**: minimum 200 multi-turn scenarios per agent. Review all flagged conversations.

3

Property-Based Input Fuzzing

- Tool: Hypothesis (Python). Generate random inputs that satisfy structural constraints. Fuzz user prompts, tool API responses, inter-agent payloads, and memory retrieval results.
- **Assert**: the system never crashes, never returns unhandled exceptions, and never exposes internal state

4

Convert Findings to Regression Tests

Every vulnerability or failure mode discovered in steps 1 through 3 is immediately codified as a deterministic regression test case. Add it to the CI/CD pipeline so the issue can never regress silently

IN THE FIELD

What we see in practice: automated scanners catch the attacks everyone already knows about. What actually breaks systems in production is more ordinary: a support ticket pasted into a chat window that happens to look like a system prompt. Fuzzing needs to cover that kind of accidental chaos, not just deliberate red-team scripts.

07

Phase 5 : Resilience & Chaos Testing

Prove the system degrades gracefully when components slow down, fail, or disappear entirely

1	**Inject Network Faults** • Tool: Toxiproxy. Place it between agents and their external dependencies. • Inject latency (5s, 15s, 30s delays on tool API calls), bandwidth throttling (1KB/s on LLM API responses), and connection resets (drop connections after a partial response). • Assert: timeout triggers retry, circuit breaker opens at threshold, fallback activates, no crash.
2	**Kill Dependencies** • Tool: Chaos Toolkit. Randomly terminate processes or containers: the vector DB, tool API server, secondary LLM provider, message queue. • Assert: the system returns partial results with clear user messaging, no data corruption, and recovery within 60 seconds of dependency restoration.
3	**LLM Failover Test** • Block the primary LLM endpoint. Assert the system fails over to a secondary model within the configured timeout, and validate that output quality on the fallback model remains above minimum thresholds.

IN THE FIELD

What we see in practice: teams that skip chaos testing usually have a retry policy. They just don't know if it's a good one until the first real outage, when retries without backoff turn a slow dependency into a self-inflicted traffic spike.

08

Phase 6: Performance, Load & Soak Testing

Prove the system holds its latency, throughput, and stability targets under real-world traffic

Step-by-Step Procedure

1 — Define User Behavior Classes

- Create a Locust TaskSet that models real user workflows. Weight simple queries 3x heavier than complex multi-agent workflows

2 — Configure Distributed Workers

- For more than 1,000 concurrent users, deploy Locust in distributed mode with one master and N workers. Command: locust -f agent_load.py --master --expect-workers=4

3 — Run Progressive Ramp

- Start at 10 users, ramp up 10 users per minute until target load, then hold for the test duration

4 — Monitor & Record

- During the test, monitor Grafana dashboards (P50/P95/P99 latency, error rate, queue depth, CPU/memory).
- Tools: Locust (load generation), Prometheus + Grafana (metrics), Jaeger (traces), py-spy (memory profiling for soak).

Load Test Scenarios

SCENARIO	DESCRIPTION	DURATION	PASS CRITERION
Baseline	10% peak users	30 min	P95 < 5s, errors < 0.1%
Normal Peak	Expected peak traffic	60 min	P95 < 8s, errors < 0.5%
Stress	150% of peak	30 min	P95 < 15s, errors < 2%
Spike	10x traffic in 30 sec	10 min	Recovers within 2 min
Soak	60% peak sustained	8–24 hrs	No drift > 20% from baseline
Breakpoint	Ramp until failure	Until break	Identify saturation point

IN THE FIELD

What we see in practice: most load tests model a single query in, a single answer out. Multi-agent workloads don't fail that way. They fail when ten planning loops queue up behind one slow tool call, and the latency curve falls apart even though no individual request was slow

09

Phase 7: Guardrail & Safety Validation

Ensure all outputs comply with safety, privacy, and organizational policy standards

Step-by-Step Procedure

1

PII Detection & Leakage Testing

- Tool: Presidio (Microsoft). Scan all agent outputs for PII patterns. Inject 100 synthetic records with names, SSNs, emails, and credit cards into the agent's data context.
- Assert: zero PII fields appear verbatim in any response; the agent uses [REDACTED] or aggregated summaries.

2

Toxicity & Harmful Content Screening

- Tool: Perspective API (Google).
- Score every agent output for toxicity.
- Threshold: toxicity score under 0.3 for all outputs; flag for human review above 0.2.

3

Hallucination Detection

- Tool: TruLens / DeepEval.
- Measure the faithfulness of agent responses to source data.
- For every factual claim in the output, verify it is entailed by the source documents.
- Target: hallucination rate under 2%; zero tolerance for safety-critical domains.

4

Prompt Injection Resistance (Production Layer)

- Tool: Promptfoo + AWS Guardrails / Azure AI Content Safety.
- Deploy input/output guardrail middleware in production.
- Continuously scan all traffic for injection patterns.
- Assert: 100% of known injection patterns blocked; refresh red-team coverage monthly

IN THE FIELD

What we see in practice: guardrails validated once before launch create a false sense of safety. Every model update, prompt change, and new tool integration is a fresh chance for a previously blocked pattern to get through. The teams that stay safe treat this phase as continuous, not a one-time gate.

10

Decision Trees: Choosing the Right Test

Not sure which test to run? Work through the questions below: they'll route you to the right phase in under a minute.

Which Test Type Do I Need?

1	**Is this a change to a single agent's prompt, tools, or output schema?** Run Phase 1: Agent Unit & Prompt Testing. Re-run the evaluation dataset.
2	**Does the change affect how agents interact or handoff data?** Run Phase 2: Workflow & Integration Testing. Validate handoff contracts and macro outcomes.
3	**Does the change affect RAG, embeddings, vector DB, or context management?** Run Phase 3: Context & Memory Validation. Benchmark retrieval accuracy and staleness.
4	**Is this a pre-release or quarterly security review?** Run Phase 4 + Phase 7: full adversarial fuzzing and guardrail validation.
5	**Are you deploying to a new environment or scaling infrastructure?** Run Phase 5 + Phase 6: resilience, load, and soak testing. If none of the above apply, run the full Phase 1–7 suite: a major change has been detected

Defect Triage Decision Framework

SYMPTOM	LIKELY ROOT CAUSE	FIRST TEST TO RUN	ESCALATION
Wrong output format	Prompt regression or schema change	Phase 1: prompt regression suite	Dev → prompt review
Incorrect facts in output	Hallucination or stale memory	Phase 3: retrieval accuracy + Phase 7: faithfulness	QA → red team
Pipeline produces no output	Handoff failure or agent crash	Phase 2: trace-based integration test	Dev → trace analysis
Slow response times	LLM latency or queue backlog	Phase 6: latency breakdown via tracing	Dev → infra team
Offensive or harmful output	Guardrail bypass	Phase 7: toxicity + injection test	QA → immediate hotfix
Intermittent failures	Resource exhaustion or race condition	Phase 5: chaos test + Phase 6: soak	Dev → memory profiling

11

Checklists & Templates

These are the same checklists our teams run on real engagements. Copy them, customize them, or drop them straight into your project management tool.

Sprint Testing Checklist (Per Sprint)

- All new/modified agents have updated evaluation datasets (at least 50 examples)
- Prompt regression suite passes at 95%+ accuracy (no drop over 2% vs. baseline)
- All agent-to-agent handoff contracts are validated
- Integration tests pass for all affected workflow pipelines
- Adversarial smoke test run (top 20 prompt injection patterns)
- PII leakage scan passes on all agent outputs
- Toxicity scan passes (score under 0.3 for all test outputs)
- All new failure modes from the sprint are added to the regression suite
- Test results documented and linked to sprint stories
- QA sign-off recorded for each completed story

Release Readiness Gate (Pre-Production)

- All Phase 1–7 test suites pass at required thresholds
- Load test completed: P95 under 8s at projected peak, error rate under 0.5%
- Soak test completed: no metric drift over 20% across 8+ hours
- Resilience test passed: system recovers from all injected failures
- Full guardrail validation suite: 100% pass rate

- Red team session completed (200+ adversarial scenarios)
- Hallucination rate confirmed under 2% on the evaluation dataset
- Prompt injection resistance: 100% of known patterns blocked
- All critical/high defects resolved; no deferred P1 bugs
- PO sign-off on quality report and risk summary
- Rollback plan documented and tested
- Monitoring dashboards and alerts configured for production

Red Team Session Checklist

- Session goal and scope defined (which agents, which attack categories)
- Participants briefed: QA lead, security engineer, AI engineer, external red teamer (optional)
- Automated scanner run first (Garak / PyRIT); results reviewed
- Manual adversarial testing: prompt injection, jailbreak, PII extraction, role confusion
- Multi-turn attack scenarios: trust escalation, context manipulation, goal hijacking
- Encoding bypass attempts: Base64, ROT13, Unicode homoglyphs, RTL override
- All findings documented with severity, reproduction steps, and recommended fix
- Critical findings escalated immediately to Dev and PO
- All findings converted to regression test cases within 48 hours
- Post-session retrospective completed

New Agent Onboarding (Before First Deployment)

- Agent contract documented: input schema, output schema, system prompt, tool access list
- Evaluation dataset created (50+ examples: happy path, edge cases, adversarial)
- Automated evaluators configured (schema validation, semantic similarity, rubric)

- Prompt regression baseline established (3–5 runs, variance measured)
- Integration points documented: upstream/downstream agents, tool dependencies
- Guardrail configuration applied: PII filter, toxicity filter, output scope limiter
- Agent added to the load test user behavior model
- Monitoring instrumented: OpenTelemetry traces, latency metrics, error counters
- Code review completed with testability as an explicit criterion
- QA sign-off: agent approved for workflow integration testing

12

Tool Quick-Start Guides

A tool only earns its place once it's running. These condensed guides take each core tool in the framework from zero to first useful result, fast.

LLM EVALUATION & TRACING

PURPOSE

End-to-end LLM tracing, dataset management, automated evaluation pipeline

INSTALL

pip install langsmith && export LANGCHAIN_API_KEY=<your-key>

KEY FEATURES

Trace every LLM call with latency/ token stats; create evaluation datasets; run evaluators (exact match, semantic similarity, custom); CI/CD integration via CLI

USE IN PHASE

Phase 1 (prompt regression),
Phase 2 (workflow tracing),
Phase 3 (memory evaluation)

QUICK START

langsmith evaluate --dataset my_eval_set --evaluator semantic_similarity --model gpt-4

Our take:

A strong default if you're already in the LangChain ecosystem. Teams on a different stack sometimes find its tracing model assumes more about your architecture than they'd like.

garak

LLM VULNERABILITY SCANNER

PURPOSE

Automated adversarial testing: prompt injection, jailbreak, data leakage, hallucination probing

INSTALL

pip install garak

KEY FEATURES

200+ built-in attack probes; supports OpenAI, Anthropic, Hugging Face, custom endpoints; generates HTML reports

USE IN PHASE

Phase 4 (adversarial fuzzing), Phase 7 (guardrail validation)

QUICK START

garak --model_type openai --model_name gpt-4 --probes all

Our take:

An excellent first pass, but its probe library goes stale as fast as jailbreak techniques evolve. A clean Garak run is a floor, not a ceiling.

ragas

RAG EVALUATION FRAMEWORK

PURPOSE

Evaluate retrieval-augmented generation quality: faithfulness, answer relevancy, context precision/ recall

INSTALL

pip install ragas

KEY FEATURES

Faithfulness (is the answer grounded in context?), answer relevancy (does it address the question?), context precision/recall (are the right docs retrieved?)

USE IN PHASE

Phase 3 (context and memory validation)

QUICK START

from ragas import evaluate; results = evaluate(dataset, metrics=[faithfulness, answer_relevancy, context_precision])

Our take:

The metrics are only as good as your groundtruth dataset. Point it at synthetic data instead of a real evaluation set and you'll get numbers that look great and mean very little.

LOCUST

LOAD TESTING

PURPOSE

Python-native load testing with scriptable user behavior; distributed mode for high concurrency

INSTALL

pip install locust

KEY FEATURES

Define user workflows as Python classes; real-time web dashboard; CSV export; distributed workers

USE IN PHASE

Phase 6 (load, stress, spike, soak testing)

QUICK START

locust -f agent_load.py -- host=https://api.example.com -- users=100 -- spawn-rate=10

Our take:

The easiest way to load test in Python, but its default user-behavior model doesn't capture bursty, cascading agent-to-agent calls well. Plan to hand-roll a task set that reflects real agent chains, not single request/response pairs.

Toxiproxy

NETWORK FAULT INJECTION

PURPOSE

Simulate network faults between services: latency, bandwidth throttle, connection drops, timeouts

INSTALL

docker run -p 8474:8474 shopify/toxiproxy(or: brew install toxiproxy)

KEY FEATURES

Create proxies per dependency; add/remove toxics (latency, bandwidth, timeout) via API or CLI; zero code changes required

USE IN PHASE

Phase 5 (resilience and chaos testing)

QUICK START

toxiproxy-cli create llm_api -l 0.0.0.0:5555 -u api.openai.com:443 && toxiproxy-cli toxic add llm_api -t latency -a latency=5000

Our take:

It does one job and stays out of the way. The real value isn't the tool, it's that most teams don't write resilience assertions until they have a way to actually inject failure. Once they do, they rarely go back to testing without it.

PRESIDIO®

PII DETECTION

PURPOSE

Detect and anonymize PII (names, SSNs, emails, credit cards, medical records) in text

INSTALL

pip install presidio-analyzer presidioanonymizer

KEY FEATURES

40+ built-in PII recognizers; NLP + regex hybrid detection; customizable entity types; anonymization operators (redact, hash, replace)

USE IN PHASE

Phase 7 (guardrail validation, PII leakage testing)

QUICK START

```
from presidio_analyzer import AnalyzerEngine; analyzer = AnalyzerEngine(); results = analyzer.analyze(text=agent_output, language='en')
```

Our take:

Strong at the entity types it knows out of the box. It won't catch domain-specific sensitive data, internal project codenames, proprietary identifiers, without extending its recognizers first. Don't treat it as a complete PII gate on day one.

13

Engagement Model: Working with Innocito

Every organization starts this journey from a different place, so we've built engagement models to match: whether you need a quick assessment, a full framework build-out, or an embedded team that owns AI quality end to end.

Assessment & Roadmap

A 2 to 4 week evaluation of your current AI testing maturity. Deliverable: gap analysis, prioritized roadmap, tool recommendations, and estimated effort.

Framework Implementation

End-to-end implementation of the seven-phase testing framework. Includes tool setup, CI/CD integration, evaluation dataset creation, and knowledge transfer.

Managed QE Service

A dedicated Innocito QE team embedded in your sprints. Continuous testing, redteaming, load testing, and quality reporting. Monthly cadence.

Red Team as a Service

Periodic adversarial testing campaigns. Our security-focused AI testers probe your system with the latest attack techniques. Quarterly or on demand.

Training & Enablement

Hands-on workshops for your QA, Dev, and BA teams. Covers AI testing fundamentals, tool proficiency, and framework adoption. One to three day sessions.

About Innocito

AI-Native Digital Engineering for Scalable, High-Performance Platforms

Design, modernize, and operate intelligent platforms built to scale, perform and evolve.

In a digital world shaped by rapid change and rising expectations, organizations need engineering partners who can design systems that scale, perform and evolve without increasing cost or operational risk. Today, digital platforms must go beyond functional software. They must be resilient, adaptive and capable of continuous learning.

Innocito is an AI-first digital engineering services company focused on building and modernizing software products and intelligent platforms engineered for scale, performance and long-term value. We combine deep engineering discipline, AI-ready and data-driven architectures, and quality-led execution to convert digital strategy into reliable, production-ready systems.

Our capabilities span product and platform engineering, quality engineering, cloud and DevOps, data and analytics, and applied AI. Across the full digital lifecycle, we embed intelligence into platforms, workflows and delivery pipelines to enable automation, insight and continuous improvement.

Our teams work alongside client teams to design systems that are reliable, adaptable and ready for intelligent operations. Quality engineering is foundational to our approach, ensuring platforms remain stable, secure and compliant as they evolve and scale.

Ready to Put This Playbook to Work?

Tell us where your AI testing stands today, and we'll map the fastest path to where it needs to be.

Book a consultation with our AI Quality Engineering team.

Services

Digital Engineering

Quality Engineering

Technology Advisory

Quick Links

About Us

Services

Resources

Careers

Contact us

United States

Dallas

Innocito Technologies llc, 511 E John Carpenter Fwy, suite 500, Irving, TX 75062, USA.

India

Hyderabad

Innocito Private Limited, 3rd Floor, The Business Park by Pranava Group, Landmark Residency, Kondapur, Hyderabad, Telangana India-500084

India

Visakhapatnam

Innocito Private Limited, Tech Mahindra Campus, Vizag City Center, Tower 2, Survey No. 44P, Old Resapuvani Palem, Visakhapatnam, Andhra Pradesh, India–530013

info@innocito.com

AI Quality Engineering (Testing AI)

Follow Us