DocRouter Engineering

Offline Evaluation Frameworks for AI Agents

Replayable datasets, repeated trials, and structured scoring to catch regressions before you ship.

Figure A. High-level offline evaluation loop shared by the Smart Agent Kit and medical claims design.

Author	Andrei Radulescu-Banu
Organization	DocRouter
Date	August 2026
Audience	Engineering leaders, AI platform teams, and practitioners building tool-using agents

1. Executive Summary

Language-model agents are stochastic. A small prompt or tool change can silently degrade performance on cases that previously worked. Classic unit tests are insufficient: every meaningful change requires a wide, repeatable retest across representative inputs.

This white paper describes an offline evaluation architecture developed for DocRouter's Document Agent (the Smart Agent Kit) and shown to transfer to a medical insurance coverage-assessment design. The architecture rests on a shared vocabulary:

agent → dataset → tasks/tags → test run → 1..k trials → per-task evaluation → aggregates

Key contributions include: (1) a clear separation of ground-truth facts, reference solutions, and assertions so agents are not forced into a single "golden" output; (2) multi-trial evaluation with pass@k and pass^k summaries to handle stochasticity; (3) deterministic graders first, LLM judge rubrics second; (4) tags, cost tracking, and tool traces that keep evaluation cheap enough for daily engineering use; and (5) versioned run configuration for reproducible historical comparison.

The same pattern supports both configuration-coding agents and structured domain agents (claims adjudication). Offline evaluation becomes part of the product development loop rather than a one-off demo script.

2. The Problem: Evaluating Stochastic Agents

Classic software needs unit tests. Prompt-based and agent systems need them even more—and they behave differently. Because language models sample from a distribution, the same task can yield different tool sequences and different valid artifacts across runs. A one-line prompt tweak can ripple across many paths: fixing case ten may silently break cases one and two.

Consequently, meaningful changes force a wide retest across representative inputs and configurations, not a single happy-path check. Without infrastructure for that retest, teams ship on hope rather than measurement.

Core risk: Stochastic systems break silently across cases. Offline evaluation is how you catch that before you ship.

3. Offline vs Online Evaluation

Offline evaluation uses controlled, replayable datasets to measure behavior and regressions before release. You define representative cases and labeled expectations, run the agent, score results, and re-run the suite after changes.

Online evaluation measures behavior on real production traffic using telemetry, automated checks, product outcomes, and explicit user feedback. Typical setups export tool calls and traces with OpenTelemetry and collect signals such as thumbs-up/thumbs-down or short textual feedback.

Dimension	Offline	Online
Primary purpose	Regression detection before ship	Production monitoring & triage
Data	Curated, versioned datasets	Live traffic & user signals
Labels	Facts, references, assertions	Implicit outcomes + explicit feedback

Cost model	Controlled; tags limit scope	Continuous; sampled or full

DocRouter's SigAgent.AI product implements online evaluation for Claude agents; this paper focuses on the offline side of the loop.

4. The Offline Evaluation Architecture

We built the Smart Agent Kit for DocRouter's Document Agent—a coding agent that configures schemas, prompts, and tags through a multi-tool agent loop. The same basic pattern applies to many other agents:

1. Define a **dataset** of representative tasks, tagged for segmentation and selective reruns.
2. Define what "good" means using **facts**, **reference solutions**, and **assertions / invariants**.
3. Run the agent offline, usually with **1..k trials per task**, recording results, traces, cost, and latency.
4. Score each trial with **deterministic graders first**, followed by **LLM judge rubrics** where semantic judgment is needed.
5. Aggregate results per task and across the test run, then rerun the full suite—or only the affected slice—after changes.

Figures 2 and 3 in the accompanying materials illustrate this shared vocabulary applied to two domains: the Smart Agent Kit (Document Agent) and a medical insurance Benefits Examiner design.

Shared architecture vocabulary

agent → **dataset** → **tasks/tags** → **test run** → **trials** → **per-task eval** → **aggregates**

5. Smart Agent Kit — Document Agent Evaluation

DocRouter's Document Agent sets up document extraction in plain language. It uses many tools: creating and validating schemas, writing prompts, attaching tags, running extraction, and more. Because this is a tool-using coding agent, chat fluency alone cannot tell us whether it worked. The Smart Agent Kit evaluates the actual outcome.

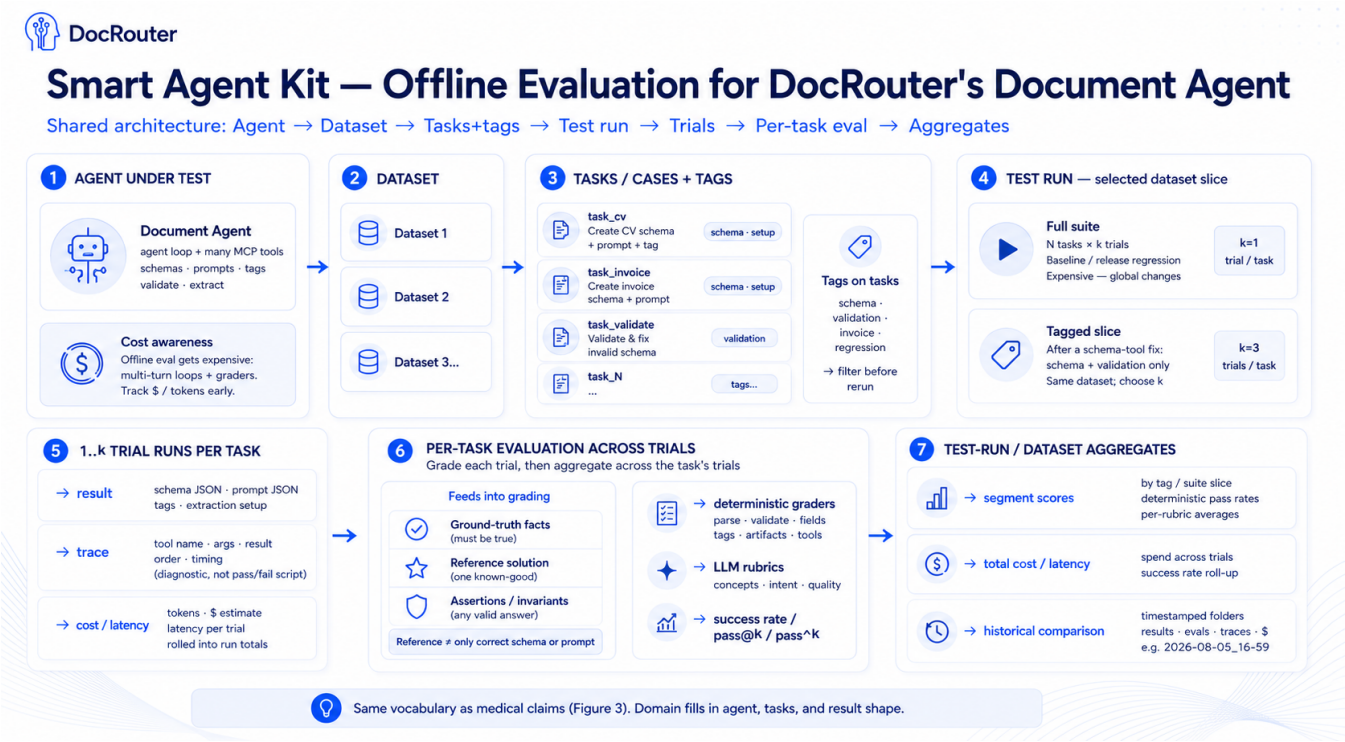

Figure 2. Smart Agent Kit — agent under test → dataset → tasks/tags → test run → 1..k trials → per-task evaluation → test-run / dataset aggregates.

Key components

- A **dataset** contains configuration tasks (create an invoice schema, build a CV extraction prompt, repair an invalid schema, …).
- Tasks carry **tags** (`schema`, `validation`, `invoice`, `regression`) for selective reruns.
- A **test run** selects a full dataset or a tagged slice.
- Each selected task runs **1..k trials**. Every trial records resulting artifacts, the tool-call trace, cost, and latency.
- Deterministic graders and LLM rubrics score each trial; scores aggregate into a **per-task evaluation**, then into test-run and dataset metrics.

Tool traces are primarily a **diagnostic signal**. Agents can take different valid paths and produce different valid artifacts. Requiring one exact tool sequence—or one golden schema—would make evaluation unnecessarily brittle. The kit evaluates the tool-using coding agent, not PDF extraction accuracy itself.

6. Defining “Correct”: Facts, References, Assertions

Calling everything “ground truth” is a subtle trap, especially for coding and configuration agents. A task may admit several correct schemas or prompts. Treating a particular expected schema as the sole ground truth quietly becomes “the schema must look like the one we happened to write.” That penalizes valid alternatives and fights the creativity expected of a tool-using agent.

Three distinct concepts

Ground-truth facts	Objectively must be true	Required field names; valid JSON; expected tag present

Reference solution	One known-good example	A schema/prompt pair known to work—not the only allowed shape
Assertions / invariants	Any acceptable solution must satisfy	“Includes patient name and dates”; “schema validates”; required tool ran when mandated

Principle: A reference solution is not necessarily the only correct solution. Grade the **outcome**, not one particular path to that outcome.

For the Document Agent, much of the evaluation can therefore be deterministic: JSON parses, schema validation passes, required fields and tags exist, extraction executes, and required invariants hold. LLM judges handle semantic questions—whether a prompt captures the requested intent or a schema adequately covers a concept.

7. Transferring the Pattern: Medical Insurance Claims

The same architecture applies to a medical insurance Benefits Examiner agent (a design, not a shipped product). Assume the agent uses tools for policy lookup, claim details, knowledge-base retrieval, and related tasks. A trial produces structured output: benefit decisions (service line, disposition, plan citation, network, cost share, medical necessity), open facts still needed, an examiner summary, and a headline determination (payable / deny / pend / partial).

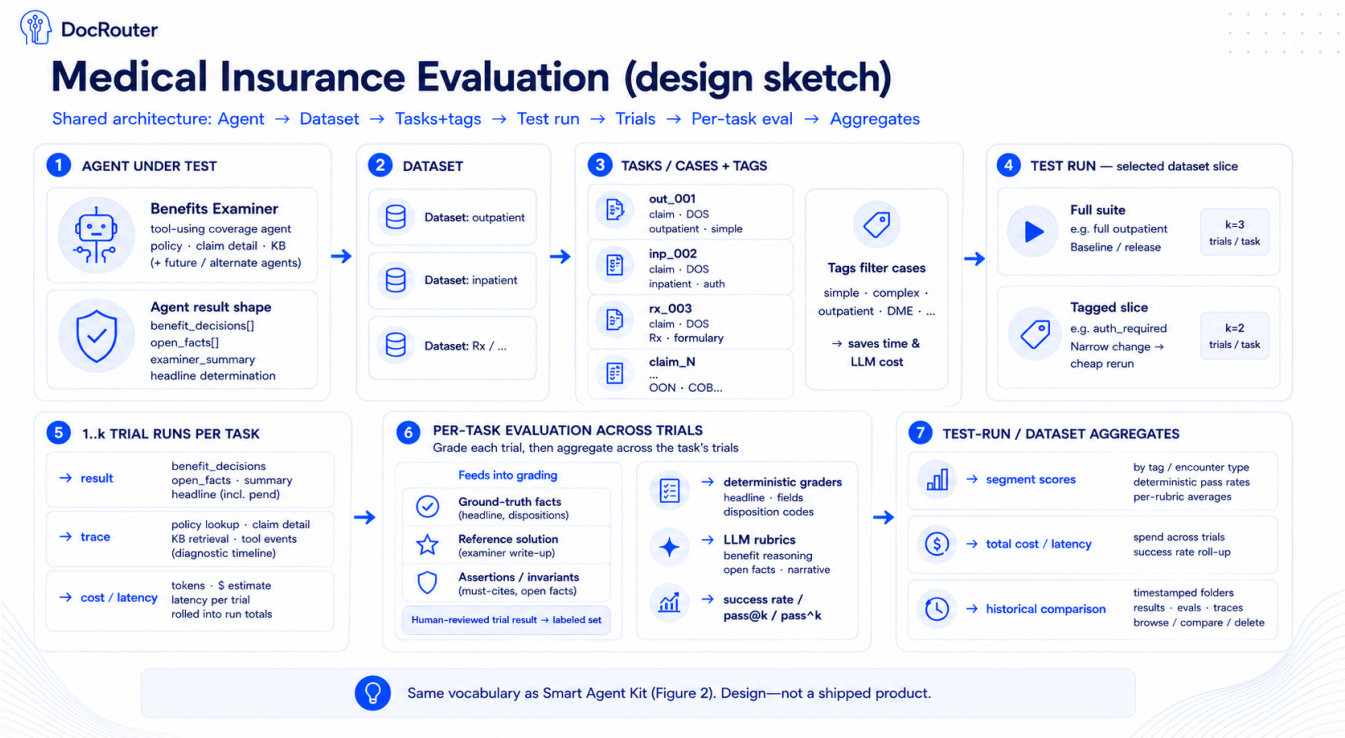

Figure 3. Medical insurance evaluation (design) — the same dataset → task → trial → evaluation pattern applied to structured coverage assessments.

Here the labeling distinction matters even more. Objective claim and policy attributes (dates, codes, eligibility, network status) can be ground-truth facts. Headline and benefit dispositions are expert-labeled decisions. An examiner write-up is a reference. Requirements such as “must cite the plan” or “must surface this open fact” are

assertions.

A strong agent result can be promoted into the labeled set, but only after domain-expert review. A high LLM-judge score should nominate a result for review, not silently turn it into tomorrow's ground truth. Offline evaluation thus serves a second purpose: it helps grow the labeled dataset safely over time.

8. Trials and Stochasticity

A single agent run is a sample, not the agent. Keep the hierarchy distinct:

Term	Meaning
Dataset	Versioned collection of tasks
Task	One test case (configuration job or claim)
Trial	One execution of the agent on that task
Test run	One suite execution over selected tasks/tags; each task runs 1..k trials
Per-task evaluation	Aggregate of that task's graded trials (success rate, mean scores, pass@k / pass^k)
Test-run aggregate	Roll-up of per-task evaluations plus cost and latency

Multiple trials matter because agents are stochastic. Two common summaries:

- **pass@k** — at least one of k trials succeeds (capability with multiple attempts)
- **pass^k** — all k trials succeed (reliability / consistency)

Running ten trials on every task quickly becomes expensive. Practical guidance: use one trial for fast development feedback; use multiple trials for release baselines and critical or high-variance tasks; tag unstable tasks and spend extra budget there.

9. Grading: Deterministic First, LLM Judges Second

"LLM-as-judge" should not mean handing the entire evaluation problem to another model. Prefer deterministic checks wherever possible: schema validation, required fields, artifact presence, expected tags, required tool use when genuinely required, assertion checks, and structural comparison only when the output truly has one canonical shape.

Then apply one or more LLM judge rubrics for semantic questions (completeness, intent match, prompt quality, domain reasoning). The pipeline is:

Deterministic checks → LLM rubric(s) → Per-task aggregate → Test-run aggregate

LLM judges should themselves be evaluated. A judge model need not be the agent model. Periodically compare judge scores against human or domain-expert judgments and adjust the rubric when they diverge. Treat each LLM judge as a scalable grader, not as ground truth.

10. Making Evaluation Cheap Enough to Use

Offline evaluation only works if engineers actually rerun it. Three features make that practical.

Tags. Segment the dataset so you can rerun only what changed. After a narrow schema-tool fix, run `schema` and `validation`; after a model or system-prompt change, run the full suite. Tags also reveal where

regressions concentrate (e.g., only on complex claims or behavioral cases).

Cost and latency. Track spend and speed as first-class metrics alongside quality. An agent that becomes slightly more accurate but three times slower or more expensive may still be a regression.

Tool traces. Scores say whether something failed; traces say how. A useful debugging loop is: failing task → inspect trial trace → fix agent or tool → retag and rerun the affected slice. Prefer outcome grading; enforce path constraints only when the path itself is the requirement.

Independent trials can also run in parallel to reduce wall-clock time without changing the evaluation methodology.

11. Reproducibility Requires Versioning

A timestamped result folder is useful history, but it is not enough. Each test run should record the versions and configuration of both the system under evaluation and the evaluator itself:

- dataset version
- selected tasks and tags
- agent / code version
- model and model configuration
- prompts and tool definitions
- grader / rubric versions
- trial count (k)

Otherwise “87 last week, 92 today” may be impossible to reproduce—or even interpret. With versioned run configuration, historical evaluation becomes a trustworthy comparison of quality, cost, latency, and reliability.

12. Conclusions and Recommendations

Offline evaluation turns agent development into engineering. The essential pattern is straightforward:

representative datasets → repeatable trials → deterministic checks → semantic grading → aggregates → traces → selective reruns

The difficult part is defining “correct” without accidentally requiring one golden output or one golden tool path. That is why the distinction between facts, reference solutions, and assertions matters: it lets an evaluator enforce what must be true while still allowing agents to find different valid solutions.

We built the Smart Agent Kit around that principle for DocRouter’s Document Agent. The medical insurance example shows that the same architecture transfers to a very different domain.

TAKEAWAY

Start labeled cases early. Measure every meaningful change. Keep traces. Track cost and latency with quality. Treat evaluation as part of the product—not a one-off demo script.

Further Reading

• Anthropic Engineering, *Demystifying evals for AI agents* — task vs trial, pass@k / pass^k, and related agent-eval practice.
https://www.anthropic.com/engineering/demystifying-evals-for-ai-agents