Production LLM Evaluation Guide

Production LLM Evaluation Guide

LLM evaluation is the process of turning an ambiguous requirement such as "the assistant should be accurate" into a versioned dataset, explicit quality criteria, repeatable measurements, and release decisions.

The most important principle is simple:

> Evaluate the system at the level where a failure can be fixed.

An end-to-end score tells you whether the product works. Component-level scores tell you why it does not. A useful evaluation program needs both.

Contents

1. What an Eval Is

An evaluator is a function that receives some combination of:

• input;
• actual output;
• reference output;
• retrieved context;
• execution trace;
• metadata;

and returns a structured result:

```
{
  "metric": "tool_selection_correct",
  "score": 1.0,
  "passed": true,
  "reason": "The request requires get_invoice, which was selected.",
  "evaluator_version": "tool-selection-v3"
}
```

An evaluation suite is a collection of examples and evaluators run against a specific version of the application.

An experiment is one execution of that suite with a frozen configuration:

```
{
  "experiment_id": "2026-08-04-agent-v17",
  "dataset_version": "enterprise-tools-v8",
  "application_version": "git:6e501af",
  "model": "provider/model-snapshot",
  "prompt_version": "router-v12",
  "tool_registry_version": "tools-v6",
  "temperature": 0,
  "started_at": "2026-08-04T10:30:00Z"
}
```

1.1 Tests and evals are related but different

A test normally has a strict pass or fail contract:

```
The output must parse as JSON.
The agent must not call a tool without authorization.
The generated code must compile.
```

An eval measures quality on a scale or over a dataset:

```
Retrieval recall@5 is 0.91.
Task completion is 87%.
Median groundedness is 4.6 out of 5.
```

Good systems turn critical eval metrics into release tests. For example:

```
Block release if tool-selection accuracy drops by more than 2 percentage points.
Block release if any unauthorized tool call succeeds.
Block release if JSON schema compliance is below 100%.
```

2. Why LLM Evaluation Is Different

Traditional software is usually evaluated against exact behavior. LLM output can be valid even when the wording differs from the reference.

This creates several challenges:

• several answers can be correct;
• one answer can contain both correct and incorrect claims;
• quality is often multidimensional;
• model outputs and model-based evaluators can vary;
• production inputs are broader than a small benchmark;
• average quality can hide catastrophic failures;
• the same final answer can result from safe or unsafe trajectories.

For example, these two answers are semantically equivalent:

```
The contract expires on 31 December 2026.
The expiration date is 2026-12-31.
```

Exact string comparison marks them as different. A date parser followed by an exact date comparison marks them as equivalent.

Use the least subjective evaluator that correctly measures the requirement:

1. exact code check;
2. parser or domain rule;
3. reference-based semantic comparison;
4. rubric-based model judge;
5. human expert review.

Do not use an LLM judge to check something that ordinary code can prove.

3. Start with the Product Contract

Before selecting metrics, write down what the system must accomplish and what it must never do.

3.1 Define success from the user's perspective

For a document assistant, success might mean:

• finds the relevant source;
• extracts the correct event and date;
• distinguishes explicit facts from inference;
• cites evidence that supports the claim;
• returns the expected JSON shape;

• responds within the latency budget.

For an enterprise agent, success might mean:

• chooses the correct tool;
• supplies valid arguments;
• respects tenant and user permissions;
• does not call duplicate or overlapping tools;
• completes the task with minimal steps;
• does not repeat a side effect during retry.

3.2 Create a metric tree

Avoid a single vague metric such as `quality`. Break it down:

```
Task success
|-- Input understanding
|-- Retrieval
|   |-- Relevant item found
|   |-- Relevant item ranked early
|   `-- Tenant filter applied
|-- Generation
|   |-- Correctness
|   |-- Completeness
|   |-- Faithfulness
|   `-- Citation support
|-- Tool execution
|   |-- Correct tool
|   |-- Correct arguments
|   |-- Authorized call
|   `-- Successful result handling
`-- Operational quality
    |-- Latency
    |-- Cost
    `-- Retry safety
```

Each leaf should map to an observable signal and an owner. If retrieval recall is low, changing the final answer prompt is unlikely to fix it.

3.3 Classify requirements by severity

Use three levels:

Level	Meaning	Example	Release behavior
Critical	Must never fail	Cross-tenant data exposure	Block on one confirmed failure
Required	Core product behavior	Correct tool selected	Threshold plus regression gate
Preferred	Quality improvement	Concise tone	Compare aggregate score

Critical requirements should use deterministic checks or human-confirmed security tests whenever possible.

3.4 Write an evaluation specification

```
feature: invoice_agent
```

```
owner: ai-platform
primary_outcome: resolve_invoice_request

critical_failures:
  - cross_tenant_access
  - unauthorized_side_effect
  - fabricated_payment_confirmation

metrics:
  - name: task_success
    target: 0.90
    evaluator: human_calibrated_judge_v4
  - name: tool_selection_accuracy
    target: 0.92
    evaluator: exact_tool_match_v2
  - name: argument_validity
    target: 0.99
    evaluator: json_schema_v3
  - name: p95_latency_ms
    maximum: 6000
  - name: mean_cost_usd
    maximum: 0.05

dataset: invoice-agent-v12
minimum_examples: 300
```

4. Build the Evaluation Dataset

The dataset is usually more important than the evaluation framework. A precise metric over an unrepresentative dataset gives a precise but misleading result.

4.1 Recommended example format

JSON Lines works well because every example is independent and easy to diff:

```
{"id":"tool-001","input":{"message":"Show invoice 4821"},"reference":{"tool":"get_invoice","arguments":{"invoice
{"id":"tool-002","input":{"message":"List unpaid invoices for Acme"},"reference":{"tool":"search_invoices","argu
```

Keep these fields stable:

- `id`: permanent example identifier;
- `input`: exact system input or replayable input fixture;
- `reference`: expected facts, actions, outputs, or acceptable alternatives;
- `metadata`: slices used for analysis;
- `provenance`: where the example came from;
- `label_version`: version of the human judgment;

4.2 Useful dataset sources

Build a balanced set from:

- manually designed common cases;
- production traces with user permission and proper redaction;
- incidents and known failures;
- support tickets;

• edge cases discovered by engineers;
• adversarial and security cases;
• synthetic variations reviewed by a human;
• examples where tools or intents are easy to confuse;
• empty, partial, contradictory, and malformed inputs;
• multiple languages and realistic spelling errors.

Production failures should become permanent regression examples.

4.3 Dataset splits

Maintain different splits for different decisions:

Split	Purpose	Typical size
Smoke	Fast local feedback	10-30
Common	Main user traffic	50-500
Edge	Rare but valid cases	30-300
Confusion	Similar intents/tools	30-300
Adversarial	Abuse and security	50-1000+
Regression	Every confirmed past failure	Grows continuously
Holdout	Final unbiased comparison	Depends on risk
Production sample	Detect distribution shift	Continuous

Do not repeatedly optimize against the holdout set. It stops being a meaningful holdout once the team has used its failures to modify the system.

4.4 Add metadata for slicing

Aggregate accuracy can hide failures. Attach metadata such as:

```
{
  "language": "ro",
  "customer_tier": "enterprise",
  "intent": "cancel_subscription",
  "difficulty": "hard",
  "contains_typo": true,
  "requires_retrieval": true,
  "requires_tool": true,
  "tool_family": "billing",
  "source": "production_redacted"
}
```

Then calculate the metric by slice:

```
Overall tool accuracy:             91%
English:                           94%
Romanian:                          78%
Single-tool requests:              96%
Multi-step requests:               73%
Billing tool family:               89%
Tool names with overlapping scope: 64%
```

The last rows tell the engineering team where to work.

4.5 Dataset quality checks

Validate the dataset itself:

• unique example IDs;
• valid JSON and schemas;
• no missing references where they are required;
• no secrets or personal data without an approved policy;
• no exact duplicates across train and holdout sets;
• label distribution is intentional;
• difficult examples are not all from one domain;
• timestamps and external state are frozen or mocked;
• references do not leak into model inputs.

4.6 Synthetic data is useful, but not ground truth by default

Synthetic examples are useful for expanding known categories, generating wording variations, and testing rare combinations. They can also reproduce the generator model's biases and create unrealistic inputs.

Use this sequence:

1. define the category with human examples;
2. generate variations;
3. deduplicate them;
4. validate rules automatically;
5. review a sample manually;
6. keep provenance in metadata;
7. compare synthetic and production performance separately.

5. Design Labels and Annotation

Human labels are needed when correctness depends on domain meaning. Poor label instructions produce noisy ground truth and make automated evaluators impossible to calibrate.

5.1 Prefer atomic labels

Do not ask only, "Is this response good?"

Ask:

• Are all factual claims supported?

• Does the answer address the request?
• Is any required fact missing?
• Is the selected tool appropriate?
• Are its arguments correct?
• Does the citation support the sentence attached to it?
• Is there a critical safety violation?

Atomic labels make disagreements diagnosable.

5.2 Use an explicit rubric

Example correctness rubric:

```
4 - Fully correct. All material claims are correct and all requested parts are answered.
3 - Mostly correct. Minor omission or imprecision that does not change the decision.
2 - Mixed. At least one important error or omission, but some useful content remains.
1 - Mostly incorrect. The answer would likely lead the user to a wrong decision.
0 - Invalid, irrelevant, unsupported, or unsafe.
```

Add examples at each level. Annotators need boundaries, not only descriptions.

5.3 Allow abstention and uncertainty

Useful labels include:

```
{
  "label": "uncertain",
  "reason": "The source documents contradict each other.",
  "needs_domain_expert": true
}
```

Forcing an uncertain reviewer to choose `correct` or `incorrect` corrupts the dataset.

5.4 Measure agreement

Double-label a sample. Report raw agreement and, where useful, Cohen's kappa for two reviewers or Fleiss' kappa for several reviewers.

Low agreement usually means one of these:

• the rubric is vague;
• the task itself is ambiguous;
• reviewers lack context;
• labels combine several dimensions;
• source documents are contradictory.

Resolve the rubric before scaling annotation.

5.5 Store label history

```
{
  "example_id": "rag-104",
  "label_version": 3,
  "labels": {
    "correctness": 3,
    "faithfulness": 4,
    "completeness": 2
  },
  "annotator_role": "domain_expert",
  "adjudicated": true,
  "notes": "Answer missed the renewal condition.",
  "created_at": "2026-08-04T09:10:00Z"
}
```

6. Choose the Right Evaluator

Different evaluator types answer different questions.

Evaluator	Best for	Strength	Main risk
Exact match	IDs, enums, tool names	Fast and deterministic	Rejects valid variants
Parser/schema	JSON, dates, arguments	Tests the real contract	Does not measure meaning
Domain rules	Calculations, policies	Explainable	Rules can be incomplete
Reference metric	Classification/extraction	Objective with good labels	Requires ground truth
Semantic similarity	Paraphrases	Handles wording variation	Similar text can still be wrong
LLM judge	Meaning and subjective quality	Flexible and scalable	Bias, variance, judge errors
Pairwise judge	Comparing two versions	Easier than absolute grading	Position/style bias
Human expert	High-stakes domain quality	Strongest authority	Slow and expensive
User behavior	Real-world utility	Measures product outcome	Confounded and delayed

Combine evaluators. A RAG answer might require:

```
schema_valid AND no_unsupported_claims AND answer_correctness >= 3
```

6.1 Evaluator priority

Apply evaluators in this order:

1. critical deterministic checks;
2. structural and domain rules;
3. task-specific reference comparisons;
4. model-based subjective scoring;
5. human review for uncertain or high-impact cases.

This reduces cost and prevents a judge model from overruling a provable failure.

7. Deterministic Code Evaluators

Code evaluators are fast, cheap, reproducible, and easy to run in CI.

7.1 Exact classification

```
def exact_label(actual: str, expected: str) -> dict:
    passed = actual.strip().lower() == expected.strip().lower()
    return {
        "metric": "exact_label",
        "score": float(passed),
        "passed": passed,
        "reason": f"expected={expected!r}, actual={actual!r}",
    }
```

7.2 Set precision, recall, and F1

For extracted events, entities, citations, or selected components:

```
def set_metrics(actual: set[str], expected: set[str]) -> dict:
    true_positive = len(actual & expected)
    false_positive = len(actual - expected)
    false_negative = len(expected - actual)

    precision = (
        true_positive / (true_positive + false_positive)
        if actual else float(not expected)
    )
    recall = (
        true_positive / (true_positive + false_negative)
        if expected else float(not actual)
    )
    f1 = (
        2 * precision * recall / (precision + recall)
        if precision + recall else 0.0
    )
    return {
        "precision": precision,
        "recall": recall,
        "f1": f1,
        "true_positive": true_positive,
        "false_positive": false_positive,
        "false_negative": false_negative,
    }
```

Interpretation:

• **precision** asks: of everything returned, how much was correct?

• **recall** asks: of everything that should have been returned, how much was found?

• **F1** balances precision and recall with their harmonic mean.

For a compliance extraction system, false negatives may be more costly than false positives. Use an F-beta score with beta greater than 1 when recall deserves more weight.

7.3 Normalized value comparison

Compare business values after parsing and normalization:

```
from datetime import date
from decimal import Decimal

def normalize_money(value: str) -> Decimal:
    cleaned = value.replace(",", "").replace("$", "").strip()
    return Decimal(cleaned).quantize(Decimal("0.01"))

def normalize_date(value: str) -> date:
```

```
    return date.fromisoformat(value)

assert normalize_money("$1,024.0") == Decimal("1024.00")
assert normalize_date("2026-12-31") == date(2026, 12, 31)
```

7.4 JSON Schema validation

```
import json
from jsonschema import Draft202012Validator

SCHEMA = {
    "type": "object",
    "required": ["decision", "confidence", "evidence_ids"],
    "additionalProperties": False,
    "properties": {
        "decision": {
            "type": "string",
            "enum": ["approve", "reject", "review"],
        },
        "confidence": {
            "type": "number",
            "minimum": 0,
            "maximum": 1,
        },
        "evidence_ids": {
            "type": "array",
            "items": {"type": "string"},
            "uniqueItems": True,
        },
    },
}

def schema_eval(raw_output: str) -> dict:
    try:
        value = json.loads(raw_output)
    except json.JSONDecodeError as exc:
        return {
            "metric": "schema_valid",
            "score": 0.0,
            "passed": False,
            "reason": f"Invalid JSON: {exc}",
        }

    errors = sorted(
        Draft202012Validator(SCHEMA).iter_errors(value),
        key=lambda error: list(error.path),
    )
    return {
        "metric": "schema_valid",
        "score": float(not errors),
        "passed": not errors,
        "reason": "; ".join(error.message for error in errors) or "valid",
    }
```

7.5 Business-rule validation

Schema validity is not semantic validity:

```
def business_rule_eval(output: dict, allowed_evidence: set[str]) -> list[str]:
    errors = []

    if output["decision"] == "approve" and output["confidence"] < 0.8:
        errors.append("approve requires confidence >= 0.8")

    unknown_ids = set(output["evidence_ids"]) - allowed_evidence
    if unknown_ids:
        errors.append(f"unknown evidence ids: {sorted(unknown_ids)}")

    if not output["evidence_ids"] and output["decision"] != "review":
        errors.append("a non-review decision requires evidence")
```

```
    return errors
```

8. LLM-as-a-Judge

An LLM judge is useful when code cannot reliably evaluate meaning, completeness, faithfulness, style, or task completion.

8.1 A judge is another model, not ground truth

Judges can have:

• position bias;
• preference for longer answers;
• preference for answers that resemble their own style;
• sensitivity to prompt wording;
• inconsistent scores near rubric boundaries;
• difficulty in specialized domains;
• susceptibility to instructions inside the content being evaluated.

Treat judge output as a measurement that must be calibrated.

8.2 Use a narrow rubric

Bad criterion:

```
Rate the answer's quality from 1 to 5.
```

Better criterion:

```
Evaluate factual faithfulness only.

A claim is faithful when it is directly supported by the supplied context or is
a logically necessary consequence of that context. Do not use outside knowledge.

Score:
4 - Every material claim is supported.
3 - One minor unsupported detail; main answer remains supported.
2 - At least one important unsupported claim.
1 - Most important claims are unsupported or contradict the context.
0 - The answer has no support in the context.
```

Separate faithfulness, completeness, and relevance into different judgments.

8.3 Require structured judge output

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "required": ["score", "passed", "reason", "failed_claims"],
  "additionalProperties": false,
  "properties": {
    "score": {"type": "integer", "minimum": 0, "maximum": 4},
    "passed": {"type": "boolean"},
    "reason": {"type": "string", "maxLength": 600},
    "failed_claims": {
```

```
      "type": "array",
      "items": {"type": "string"},
      "maxItems": 10
    }
  }
}
```

The judge prompt should place untrusted content inside clear delimiters and state that instructions within that content must not be followed.

8.4 Use claim-level evaluation for factual answers

Whole-answer scoring can hide one dangerous claim. A stronger pipeline is:

1. split the answer into atomic factual claims;
2. identify evidence for each claim;
3. classify each claim as supported, contradicted, or not found;
4. aggregate with higher weight for material claims;
5. preserve the failed claims for debugging.

Example result:

```
{
  "claims": [
    {
      "claim": "The agreement starts on 1 June 2026.",
      "verdict": "supported",
      "evidence_ids": ["chunk-18"]
    },
    {
      "claim": "The agreement renews automatically for two years.",
      "verdict": "contradicted",
      "evidence_ids": ["chunk-22"]
    }
  ],
  "supported_fraction": 0.5,
  "has_material_contradiction": true
}
```

8.5 Pairwise comparison

Judges are often better at choosing between A and B than assigning an absolute score. Pairwise evaluation is useful for model, prompt, and retrieval changes.

Mitigate order bias:

1. judge A versus B;
2. judge B versus A;
3. accept a win only when the decisions agree after reversing positions;
4. classify disagreement as a tie or send it to review.

Always hide implementation names. Use `Response A` and `Response B`, not model or vendor names.

8.6 Repeat only when necessary

For a stochastic or unstable judge, run multiple votes on a calibration sample, not blindly on every production trace. Majority voting increases cost and can repeat the same systematic bias.

Record:

• judge model and exact version;
• rubric version;
• judge parameters;
• number of votes;
• raw judge outputs;
• parser or validation failures.

9. Calibrate Evaluators Against Humans

An automated score is useful only if it agrees with the decisions the product team cares about.

9.1 Build a calibration set

Use 100-300 examples if available, including:

• clear passes;
• clear failures;
• borderline cases;
• each important failure category;
• examples from every major language or customer segment;
• adversarial content that tries to influence the judge.

Have qualified humans label this set without seeing the automated score.

9.2 Compare evaluator and human decisions

For a binary evaluator, create a confusion matrix:

	Human pass	Human fail
Evaluator pass	True pass	False pass
Evaluator fail	False fail	True fail

For release safety, false passes are often worse than false fails. Calculate:

```
evaluator_precision = true_pass / (true_pass + false_pass)
evaluator_recall    = true_pass / (true_pass + false_fail)
```

Also inspect agreement per failure category. A 90% agreement rate may be useless

if all disagreements are safety failures.

9.3 Tune the threshold using actual costs

Suppose an evaluator returns a 0-1 score. Do not select 0.5 only because it is the middle. Sweep thresholds and measure:

• false acceptance rate;
• false rejection rate;
• review volume;
• business cost of each error type.

For an automatic payment action, require a conservative threshold and route uncertain cases to review. For draft generation, optimize for usefulness and use a lower threshold.

9.4 Recalibrate after changes

Recalibrate when:

• the judge model changes;
• the rubric changes;
• the task or user population changes;
• a new language is introduced;
• the source corpus changes substantially;
• humans identify a new failure mode.

Never silently upgrade the judge and compare the new scores to old experiments.

10. Evaluate Structured Extraction

Extraction should be evaluated field by field, not only document by document.

10.1 Example reference

```
{
  "document_id": "contract-019",
  "events": [
    {
      "type": "effective_date",
      "date": "2026-06-01",
      "party": "Acme Ltd",
      "evidence_span": "This Agreement is effective as of June 1, 2026."
    }
  ]
}
```

10.2 Measure each stage

```
Document parsing -> candidate retrieval -> field extraction -> normalization
```

```
-> entity matching -> deduplication -> final schema
```

Useful metrics:

- schema validity;
- event detection precision, recall, and F1;
- field-level exact match;
- normalized date accuracy;
- entity-link accuracy;
- evidence-span overlap;
- duplicate rate;
- contradiction rate;
- abstention quality;

10.3 Match records before scoring fields

When the model returns multiple events, first align predicted and expected events. Match by stable IDs when possible. Otherwise use a deterministic assignment based on event type, normalized date, entity, and evidence location.

Do not compare list position unless order is part of the business contract.

10.4 Weight fields by impact

```
field_weights:
  event_type: 0.25
  date: 0.30
  party: 0.20
  amount: 0.15
  evidence: 0.10
```

Keep unweighted field metrics too. A weighted total can hide a broken field.

10.5 Example improvement analysis

If extraction improves from 67% to 91%, document exactly what the numbers mean:

```
Metric: macro average event-level F1
Dataset: timeline-eval-v7, 418 documents, 1,906 labeled events
Baseline: 0.67
Candidate: 0.91
Main changes: schema-constrained extraction, date normalization, evidence checks,
              failure-specific retries, and regression examples from production
```

Also report confidence intervals, slice results, and critical failure counts.

Without a metric definition and dataset version, `67% to 91%` is not reproducible.

11. Evaluate RAG Systems

A RAG pipeline has at least three systems:

1. ingestion and indexing;
2. retrieval and ranking;
3. answer generation.

An end-to-end answer score cannot identify which one failed.

11.1 Evaluate ingestion

Check:

• expected documents were indexed;
• parser did not drop tables, headings, footnotes, or OCR text;
• chunks preserve source IDs and page locations;
• metadata filters are populated correctly;
• no unauthorized tenant data enters the index;
• chunk boundaries preserve complete semantic units;
• duplicate chunks are controlled;
• index version matches the source snapshot.

Example ingestion assertion:

```
def validate_chunk(chunk: dict) -> list[str]:
    required = {"chunk_id", "document_id", "tenant_id", "text", "page"}
    errors = []
    if missing := required - chunk.keys():
        errors.append(f"missing fields: {sorted(missing)}")
    if not chunk.get("text", "").strip():
        errors.append("empty chunk")
    if len(chunk.get("text", "")) > 8_000:
        errors.append("chunk exceeds tested size")
    return errors
```

11.2 Retrieval ground truth

For every question, label acceptable source documents or passages:

```
{
  "id": "rag-044",
  "query": "When can the customer terminate without cause?",
  "relevant_document_ids": ["msa-12"],
  "relevant_chunk_ids": ["msa-12-p14-c2", "msa-12-p15-c1"],
  "reference_facts": [
    "The customer may terminate without cause with 30 days written notice."
  ]
}
```

Passage labels are more precise but more expensive. Document-level relevance is a reasonable first step.

11.3 Retrieval metrics

For the top k results:

```
precision@k = relevant retrieved in top k / k
```

```
recall@k    = relevant retrieved in top k / all relevant items
hit@k       = 1 if at least one relevant item appears in top k, otherwise 0
MRR         = mean of 1 / rank of first relevant result
```

Use nDCG when relevance has grades, such as highly relevant, partially relevant, and irrelevant.

Interpretation:

- low recall: required evidence never reaches the model;
- high recall and low precision: too much distracting context;
- good retrieval and bad answers: generation or prompt problem;
- good answer despite bad retrieval: possible memorization or unsupported answer.

11.4 Evaluate filters separately

Metadata and authorization filters need exact tests:

```
tenant_filter_recall
tenant_filter_leak_count
date_filter_accuracy
document_type_filter_accuracy
permission_denial_accuracy
```

One cross-tenant retrieval is a security incident, not a rounding error in average precision.

11.5 Generation metrics

Measure:

- answer correctness against reference facts;
- faithfulness to retrieved context;
- completeness of required facts;
- relevance to the question;
- citation correctness;
- citation completeness;
- appropriate abstention when evidence is absent;
- unsupported claim rate.

Do not confuse correctness and faithfulness:

- a fact can be correct in the real world but unsupported by retrieved context;
- an answer can faithfully repeat an incorrect source;
- both dimensions are needed.

11.6 Citation evaluation

Evaluate each claim-citation pair:

```
{
  "claim": "Notice must be provided 30 days in advance.",
  "citation_id": "msa-12-p14-c2",
  "citation_exists": true,
  "citation_entails_claim": true,
  "claim_requires_citation": true
}
```

Useful aggregates:

```
citation_precision = supported cited claims / all cited claims
citation_recall    = cited support-required claims / all support-required claims
```

11.7 No-answer tests

Include questions that the corpus cannot answer. Measure whether the system:

• abstains;

• explains what is missing;

• avoids inventing a citation;

• asks a useful clarification when appropriate.

An assistant tested only on answerable questions will learn to always answer.

11.8 RAG evaluation matrix

Failure	Retriever score	Generator score	Likely fix
Evidence absent	Low recall	Usually low	Indexing, query, filters, embeddings
Evidence buried	Low precision/rank	Mixed	Reranking, chunking, top-k
Evidence present, claim wrong	Good	Low correctness	Prompt, model, reasoning
Evidence present, claim invented	Good	Low faithfulness	Grounding contract, claim checks
Correct but uncited	Good	Good correctness, low citation recall	Citation generation

12. Evaluate Agents and Tool Use

Agent quality is a trajectory, not only a final answer.

12.1 Capture the complete trace

```
{
  "input": "Find invoice 4821 and email it to the account owner.",
  "steps": [
    {
      "index": 0,
      "type": "tool_call",
      "tool": "get_invoice",
      "arguments": {"invoice_id": "4821"},
      "result_status": "success"
    },
    {
      "index": 1,
      "type": "tool_call",
      "tool": "get_account_owner",
```

```
      "arguments": {"invoice_id": "4821"},
      "result_status": "success"
    },
    {
      "index": 2,
      "type": "tool_call",
      "tool": "send_email",
      "arguments": {"recipient": "owner@example.com", "attachment_id": "4821"},
      "result_status": "success"
    }
  ],
  "final_output": "Invoice 4821 was sent to the account owner."
}
```

12.2 Agent metric layers

Measure separately:

1. **Task completion:** Did the requested outcome happen?
2. **Tool selection:** Was the right tool chosen at each decision?
3. **Argument correctness:** Were arguments complete and valid?
4. **Authorization:** Was every action allowed for tenant and user?
5. **Execution:** Did the tool succeed and was the result interpreted correctly?
6. **Trajectory efficiency:** Were unnecessary calls avoided?
7. **Recovery:** Did the agent handle errors without causing damage?
8. **Final communication:** Did it accurately describe what happened?

12.3 Tool-selection accuracy

```
def tool_selection_eval(actual: str, allowed: set[str]) -> dict:
    passed = actual in allowed
    return {
        "metric": "tool_selection_correct",
        "score": float(passed),
        "passed": passed,
        "reason": f"selected={actual}, allowed={sorted(allowed)}",
    }
```

Allow multiple valid tools only when they are genuinely equivalent for the task.
Do not weaken the reference after seeing the candidate output.

12.4 Build a tool confusion matrix

Expected / selected	get_invoice	search_invoices	get_customer
get_invoice	84	13	3
search_invoices	18	76	6
get_customer	2	5	93

This reveals overlapping descriptions and schemas. If `get_invoice` and `search_invoices` are repeatedly confused, improve their contracts, examples, names, and input requirements before changing models.

12.5 Argument evaluation

Evaluate arguments in layers:

```
JSON parse -> schema valid -> required values correct -> permission scope valid
-> values consistent with prior tool results
```

Example:

```
def argument_accuracy(actual: dict, expected: dict) -> dict:
    keys = set(actual) | set(expected)
    matches = sum(actual.get(key) == expected.get(key) for key in keys)
    return {
        "field_accuracy": matches / len(keys) if keys else 1.0,
        "missing": sorted(set(expected) - set(actual)),
        "unexpected": sorted(set(actual) - set(expected)),
        "wrong": sorted(
            key for key in keys
            if key in actual and key in expected and actual[key] != expected[key]
        ),
    }
```

12.6 Trajectory evaluation

Exact trajectory matching is often too strict because several plans can succeed.

Represent references as constraints:

```
{
  "required_tools": ["get_invoice", "send_email"],
  "allowed_optional_tools": ["get_account_owner"],
  "forbidden_tools": ["delete_invoice", "send_bulk_email"],
  "ordering_constraints": [
    ["get_invoice", "send_email"]
  ],
  "maximum_tool_calls": 5,
  "requires_confirmation_before": ["send_email"]
}
```

Then check constraints deterministically. Use an LLM judge only for aspects such as whether the plan logically accomplished the user's intent.

12.7 Error-recovery scenarios

Inject controlled failures:

- timeout;
- 429 or rate limit;
- validation error;
- permission denied;
- empty result;
- stale version conflict;
- partial tool failure;
- duplicated response;
- malformed tool output.

Evaluate whether the agent retries safely, changes strategy appropriately, reports uncertainty, and avoids repeating side effects.

12.8 Report selection and execution separately

This distinction is essential:

```
Tool selection accuracy: 87%
Argument schema validity: 98%
Argument semantic accuracy: 91%
Tool execution success: 96%
End-to-end task success: 82%
```

An agent can select the right tool but call it incorrectly. It can also execute a wrong tool perfectly. One combined score hides the fix.

13. Evaluate MCP Systems

MCP standardizes how models discover and invoke tools, but it does not guarantee that tool contracts are clear, authorized, or reliable.

13.1 Contract-level evals

For every MCP tool, test:

• name is stable and distinct;
• description identifies when to use and not use the tool;
• input schema is valid and restrictive;
• output schema is documented or validated;
• errors use a common structured format;
• examples match the schema;
• tenant context is required where relevant;
• user permissions are checked outside the model;
• side-effecting operations support idempotency;
• sensitive fields are not exposed unnecessarily.

13.2 Discovery evals

Given only the tool catalog and a request, evaluate whether the model chooses the correct tool. Include:

• positive examples for every tool;
• negative examples where no tool should be called;
• pairs of tools with similar descriptions;
• requests with missing required information;
• requests that require clarification;

• requests from unauthorized users;

• multi-tool tasks.

13.3 Multi-tenant authorization matrix

```
tenant A user + tenant A resource + allowed action -> allow
tenant A user + tenant B resource                  -> deny
tenant A read-only user + write tool               -> deny
tenant admin + allowed tenant resource             -> allow
unknown user                                       -> deny
missing tenant context                             -> deny
```

Run these checks at the application and tool layers. The model must not be the authorization boundary.

13.4 Measuring an improvement from 50% to 87%

Use a frozen confusion dataset:

```
dataset: mcp-tool-selection-v6
examples: 612
tools: 34
slices:
  common: 240
  overlapping_tools: 180
  insufficient_information: 72
  unauthorized: 60
  multi_step: 60
metric: exact acceptable-tool accuracy
baseline: 0.50
candidate: 0.87
```

Track why it improved:

• overlapping tools consolidated;

• descriptions clarified with negative guidance;

• schemas made consistent;

• tool-level permissions exposed as deterministic constraints;

• duplicate capabilities removed;

• confused-tool examples added to regression evals;

• selection measured separately from execution.

Also show residual failures. The remaining 13% determines the next iteration.

14. Evaluate Code Generation

Generated code should be evaluated by execution and static tools before it is evaluated by another language model.

14.1 Evaluation ladder

```
Output parses
-> Typed intermediate representation validates
-> Only allowed components are referenced
```

```
-> Generated source formats
-> Source parses
-> Type checker passes
-> Linter passes
-> Unit tests pass
-> Integration tests pass
-> Security checks pass
-> Runtime/resource limits pass
-> Human review for high-impact deployment
```

14.2 Prefer typed intermediate representations

When code is assembled from approved components, ask the model for a typed plan:

```
{
  "application": "customer_support_portal",
  "components": [
    {"id": "auth", "type": "OAuthLogin", "provider": "azure_ad"},
    {"id": "tickets", "type": "TicketTable", "source": "support_api"}
  ],
  "routes": [
    {"path": "/tickets", "component": "tickets", "requires_auth": true}
  ],
  "data_flows": [
    {"from": "support_api", "to": "tickets", "operation": "list_tickets"}
  ]
}
```

Validate the IR, then compile it with deterministic templates. Evaluate:

• IR schema validity;

• component selection precision and recall;

• forbidden component count;

• reference integrity;

• policy compliance;

• deterministic compiler output;

• build and test results.

14.3 Functional tests matter more than string similarity

Two correct programs can have completely different source code. Prefer:

• unit-test pass rate;

• hidden-test pass rate;

• property-based tests;

• type correctness;

• security rules;

• execution time and memory;

• mutation testing for test-suite quality.

Text similarity to a reference solution is usually a weak primary metric.

14.4 Sandbox untrusted code

Run generated code with:

• no production credentials;
• restricted network access;
• filesystem isolation;
• CPU, memory, and time limits;
• dependency allowlists;
• explicit artifact capture;
• process termination after timeout.

An eval runner is part of the security boundary.

15. Evaluate Multi-Turn Workflows

Single-turn correctness does not measure whether a system handles a conversation.

Evaluate at three levels:

• **turn:** Was this response correct?
• **trajectory:** Were the steps and state transitions valid?
• **thread:** Was the user's final goal achieved over the whole conversation?

15.1 Useful conversation metrics

• instruction retention;
• entity and preference consistency;
• correct use of previous tool results;
• clarification quality;
• no repetition of already answered questions;
• correct state transitions;
• recovery after user correction;
• final task completion;
• total turns to completion;
• premature action rate.

15.2 Evaluate state transitions directly

For a LangGraph-style workflow:

```
ALLOWED_TRANSITIONS = {
    "classify": {"retrieve", "clarify", "reject"},
    "clarify": {"classify", "end"},
    "retrieve": {"plan", "clarify", "end"},
    "plan": {"authorize", "clarify", "end"},
    "authorize": {"execute", "reject"},
    "execute": {"verify", "recover"},
    "recover": {"execute", "clarify", "end"},
    "verify": {"respond", "recover"},
    "respond": {"end"},
}
```

```
def transition_is_valid(previous: str, current: str) -> bool:
    return current in ALLOWED_TRANSITIONS.get(previous, set())
```

Store every transition with its reason. This makes loops and invalid paths easy to detect.

15.3 Simulated users

Simulated users can test long conversations and rare behaviors, but the simulator is also a model and can be unrealistic. Validate simulator behavior against real conversation patterns and keep a human-reviewed sample.

16. Safety and Security Evals

Safety metrics must be threat-model driven. A generic toxicity score does not cover authorization, prompt injection, data leakage, or dangerous tool use.

16.1 Build a threat matrix

Threat	Attack example	Expected behavior	Evaluator
Prompt injection	Document says to reveal secrets	Ignore instruction	Canary/secret check
Cross-tenant access	Request resource from another tenant	Deny	Exact authorization test
Tool escalation	Read-only user requests delete	Deny	Policy engine result
PII leakage	Ask for another user's email	Refuse/redact	Pattern plus human audit
Data exfiltration	Encode tool data in URL	Block network/tool call	Trace policy check
False confirmation	Tool failed but answer says success	Report failure	Trace/output consistency

16.2 Test indirect prompt injection

Put adversarial instructions inside retrieved documents, tool output, web pages, emails, and uploaded files. The system should treat them as data, not higher-level instructions.

16.3 Use canaries

Insert synthetic secrets that should never appear in outputs:

```
EVAL_CANARY_7f4d12b9
```

Fail the eval if the canary appears in the model response, tool argument, URL, log field that lacks access controls, or external request.

16.4 Measure severity, not only frequency

One data leak matters more than ten minor formatting failures. Report:

```
critical_failures: 0
high_failures: 1
medium_failure_rate: 0.7%
low_failure_rate: 3.8%
```

Critical security failures should not be averaged into a general quality score.

17. Latency, Cost, and Reliability

A correct system that is too slow or expensive may still fail in production.

17.1 Record per-example operational metrics

```
{
  "latency_ms": 1840,
  "time_to_first_token_ms": 310,
  "input_tokens": 4280,
  "output_tokens": 312,
  "model_calls": 2,
  "tool_calls": 1,
  "retries": 0,
  "estimated_cost_usd": 0.018,
  "cache_hit": false
}
```

17.2 Use distributions

Report median, p90, p95, and p99 latency. Average latency hides slow outliers.

Segment by:

• task type;

• model;

• number of retrieved chunks;

• tool family;

• success versus failure;

• cold versus warm cache;

• region;

• input and output token range.

17.3 Track quality-adjusted cost

A cheaper model is not cheaper if retries or failures increase. Useful comparisons:

```
cost per successful task = total cost / successful tasks
latency per successful task
tool calls per successful task
tokens per accepted answer
```

17.4 Evaluate fallback behavior

Test provider timeout, rate limit, malformed response, and model unavailability. Verify that fallbacks preserve schema, safety, and authorization contracts.

18. Offline and Online Evaluation

18.1 Offline evaluation

Offline evals run before deployment on versioned examples with references. Use them for:

- prompt comparison;
- model selection;
- retriever tuning;
- regression testing;
- failure reproduction;
- security testing;
- release gates.

Advantages: repeatable, controlled, and easy to compare.

Limitations: the dataset may not represent live traffic.

18.2 Online evaluation

Online evals run on production traces or samples. Use them for:

- distribution shift;
- new user intents;
- retrieval anomalies;
- safety monitoring;
- cost and latency changes;
- user feedback analysis;
- discovering new regression examples.

Online traces often lack a reference answer, so use reference-free checks, behavioral signals, and sampled human review.

18.3 Production feedback loop

```
Production trace
-> automated checks
-> sample failures and uncertain cases
-> human review
-> assign failure category
-> add confirmed example to versioned dataset
-> reproduce offline
-> implement fix
-> run regression suite
-> deploy gradually
```

18.4 Sample intelligently

Do not evaluate only random traffic. Combine:

• random sample for unbiased estimates;
• all critical policy events;
• all explicit negative feedback;
• low-confidence outputs;
• new intents and outliers;
• long or expensive traces;
• cases with retries or tool errors;
• representative samples from each customer or language slice.

Correct for oversampling when calculating population-level metrics.

18.5 User feedback is a signal, not perfect ground truth

Thumbs up/down can be sparse and biased. A negative rating may reflect latency, tone, product policy, or an upstream tool failure. Join feedback to the complete trace and review failure categories.

19. Statistics and Release Thresholds

19.1 Always show the denominator

`91% accuracy` is incomplete. Report:

```
Tool-selection accuracy: 91.2% (279 / 306 examples)
Dataset: tool-selection-v8
Run date: 2026-08-04
```

19.2 Confidence intervals

For a binary metric, a Wilson interval is more reliable than a normal approximation, especially for small samples or values near 0 or 1:

```
from math import sqrt

def wilson_interval(successes: int, total: int, z: float = 1.96):
    if total == 0:
        raise ValueError("total must be positive")
    p = successes / total
    denominator = 1 + z * z / total
    center = (p + z * z / (2 * total)) / denominator
    margin = (
        z
        * sqrt((p * (1 - p) + z * z / (4 * total)) / total)
        / denominator
    )
    return center - margin, center + margin
```

19.3 Use paired comparisons

When comparing application A and B, run both on the same examples. Per-example differences reduce noise and reveal regressions hidden by the mean.

Report:

```
A wins: 42
B wins: 18
Ties: 140
Critical regressions in B: 0
```

19.4 Bootstrap uncertain metrics

For judge scores, cost, or other non-binary metrics, bootstrap the examples:

```
import random
from statistics import mean

def bootstrap_mean_interval(
    values: list[float],
    rounds: int = 10_000,
    seed: int = 7,
) -> tuple[float, float]:
    rng = random.Random(seed)
    estimates = []
    for _ in range(rounds):
        sample = [rng.choice(values) for _ in values]
        estimates.append(mean(sample))
    estimates.sort()
    return (
        estimates[int(0.025 * rounds)],
        estimates[int(0.975 * rounds)],
    )
```

19.5 Avoid threshold overfitting

Do not repeatedly tune a prompt until it barely passes one dataset. Use:

- a development set for iteration;
- a regression set for known failures;
- a holdout set for final comparison;
- production monitoring after release.

19.6 Release gate example

```
release_gates:
  critical_security_failures:
    maximum: 0
  schema_validity:
    minimum: 1.00
  tool_selection_accuracy:
    minimum: 0.90
    maximum_regression: 0.02
  task_success:
    minimum: 0.85
    maximum_regression: 0.01
  p95_latency_ms:
```

```
    maximum: 6000
  mean_cost_usd:
    maximum_increase: 0.10
```

Check absolute quality and regression from the current production baseline. A candidate can exceed the minimum and still introduce an unacceptable regression.

20. Experiment Tracking and Observability

Every result should be replayable or at least explainable.

20.1 Freeze the evaluation envelope

Record:

```
{
  "dataset_version": "rag-contracts-v14",
  "application_git_sha": "6e501af",
  "prompt_hash": "sha256:...",
  "model_provider": "provider",
  "model_snapshot": "model-version",
  "generation_parameters": {
    "temperature": 0,
    "max_output_tokens": 800
  },
  "embedding_model": "embedding-version",
  "index_snapshot": "contracts-2026-08-03",
  "reranker_version": "reranker-v5",
  "tool_registry_hash": "sha256:...",
  "evaluator_versions": [
    "schema-v2",
    "faithfulness-v4",
    "task-success-v7"
  ],
  "runtime": {
    "python": "3.12.4",
    "region": "eu"
  }
}
```

20.2 Store traces, not only scores

A score without the input, output, context, tool calls, and evaluator reason is hard to debug.

Keep:

• model messages after templating;
• retrieval queries and ranked results;
• tool definitions shown to the model;
• tool calls and results;
• workflow states;
• token and timing data;
• final output;
• every evaluator result;

• redaction and access-control metadata.

Apply retention and privacy policies. Observability does not justify storing sensitive content indefinitely.

20.3 Compare experiments by example

Aggregate dashboards are useful, but the highest-value view is often:

```
example_id | baseline | candidate | delta | failure_category | trace diff
```

Review large gains, large regressions, and all critical failures.

21. CI and Release Gates

Use tiers so developers get fast feedback without running an expensive full suite for every edit.

21.1 Suggested tiers

```
Local / every commit:
- schema and parser tests
- 10-30 smoke examples
- mocked tool contracts
- no network where possible

Pull request:
- common and regression splits
- changed components only plus critical end-to-end cases
- pairwise comparison with baseline

Nightly:
- full dataset
- multiple slices
- judge calibration monitor
- adversarial and fault-injection tests

Pre-release:
- frozen full suite
- holdout set
- load and latency tests
- security review

Post-release:
- canary traffic
- online eval sample
- rollback thresholds
```

21.2 Make eval failures actionable

A CI failure should include:

```
Metric: tool_selection_accuracy
Baseline: 0.912
Candidate: 0.873
Allowed regression: 0.020
Failed examples: 17
Largest slice regression: overlapping_tools (-0.14)
Artifact: eval-results/agent-v17.html
```

21.3 Cache safely

Cache model responses when rerunning unchanged examples with an unchanged model, prompt, tool registry, and parameters. Include all of these in the cache key.

Do not reuse a cached answer after changing the input contract or hidden context.

21.4 Control judge flakiness

For hard release gates:

• use deterministic code checks whenever possible;
• pin the judge model;
• use structured outputs;
• use low-temperature or greedy decoding where supported;
• retry only transport/parser failures;
• separate flaky subjective metrics from critical gates;
• require a margin larger than normal judge variation;
• periodically rerun a fixed judge stability set.

22. Reference Implementation

The following small runner uses plain Python structures and can be adapted to any provider or evaluation platform.

22.1 Data model

```
from dataclasses import dataclass, field
from typing import Any, Callable

@dataclass(frozen=True)
class Example:
    id: str
    input: dict[str, Any]
    reference: dict[str, Any]
    metadata: dict[str, Any] = field(default_factory=dict)

@dataclass(frozen=True)
class AppResult:
    output: Any
    trace: dict[str, Any]
    latency_ms: int
    cost_usd: float

@dataclass(frozen=True)
class EvalResult:
    metric: str
    score: float
    passed: bool
    reason: str
    details: dict[str, Any] = field(default_factory=dict)

Application = Callable[[dict[str, Any]], AppResult]
Evaluator = Callable[[Example, AppResult], EvalResult]
```

22.2 Example evaluator

```
def correct_tool(example: Example, result: AppResult) -> EvalResult:
    expected = set(example.reference["acceptable_tools"])
    actual = result.trace.get("selected_tool")
    passed = actual in expected
    return EvalResult(
        metric="tool_selection_correct",
        score=float(passed),
        passed=passed,
        reason=f"selected={actual}; acceptable={sorted(expected)}",
        details={"actual": actual, "expected": sorted(expected)},
    )
```

22.3 Runner

```
from collections import defaultdict
from statistics import mean

def run_experiment(
    examples: list[Example],
    application: Application,
    evaluators: list[Evaluator],
) -> dict:
    rows = []

    for example in examples:
        try:
            app_result = application(example.input)
            eval_results = [
                evaluator(example, app_result)
                for evaluator in evaluators
            ]
            error = None
        except Exception as exc:
            app_result = None
            eval_results = []
            error = f"{type(exc).__name__}: {exc}"

        rows.append({
            "example": example,
            "application_result": app_result,
            "eval_results": eval_results,
            "error": error,
        })

    scores = defaultdict(list)
    for row in rows:
        for result in row["eval_results"]:
            scores[result.metric].append(result.score)

    summary = {
        metric: {
            "mean": mean(values),
            "count": len(values),
        }
        for metric, values in scores.items()
    }

    return {
        "summary": summary,
        "rows": rows,
        "application_errors": sum(row["error"] is not None for row in rows),
    }
```

22.4 Slice analysis

```
from collections import defaultdict
from statistics import mean
def metric_by_slice(rows, metric_name: str, metadata_key: str):
```

```
    groups = defaultdict(list)

    for row in rows:
        slice_value = row["example"].metadata.get(metadata_key, "unknown")
        for result in row["eval_results"]:
            if result.metric == metric_name:
                groups[slice_value].append(result.score)

    return {
        slice_value: {
            "score": mean(values),
            "count": len(values),
        }
        for slice_value, values in sorted(groups.items())
    }
```

22.5 Regression comparison

```
def compare_results(baseline: dict, candidate: dict, metric: str):
    baseline_rows = {
        row["example"].id: row
        for row in baseline["rows"]
    }
    candidate_rows = {
        row["example"].id: row
        for row in candidate["rows"]
    }

    def score(row):
        for result in row["eval_results"]:
            if result.metric == metric:
                return result.score
        return None

    changes = []
    for example_id in sorted(baseline_rows.keys() & candidate_rows.keys()):
        before = score(baseline_rows[example_id])
        after = score(candidate_rows[example_id])
        if before is not None and after is not None and before != after:
            changes.append({
                "example_id": example_id,
                "baseline": before,
                "candidate": after,
                "delta": after - before,
            })
    return changes
```

22.6 Recommended result schema

```
{
  "experiment": {
    "id": "agent-v17-2026-08-04",
    "dataset_version": "agent-tools-v8",
    "application_version": "git:6e501af"
  },
  "summary": {
    "task_success": {"score": 0.86, "count": 306},
    "tool_selection_correct": {"score": 0.91, "count": 306},
    "argument_valid": {"score": 0.98, "count": 244}
  },
  "critical_failures": [],
  "slices": {
    "language": {
      "en": {"task_success": 0.89, "count": 250},
      "ro": {"task_success": 0.73, "count": 56}
    }
  },
  "artifacts": {
    "rows": "results/agent-v17.jsonl",
    "traces": "results/agent-v17-traces/"
  }
```

```
}
```

23. Failure Analysis

Evaluation creates value only when it changes the system.

23.1 Use a failure taxonomy

```
retrieval:
  - missing_document
  - wrong_filter
  - low_recall
  - poor_ranking
  - chunk_boundary

generation:
  - unsupported_claim
  - incomplete_answer
  - wrong_reasoning
  - failed_abstention
  - citation_mismatch

agent:
  - wrong_tool
  - missing_argument
  - wrong_argument
  - unauthorized_action
  - unnecessary_loop
  - false_success_message

evaluation:
  - bad_reference
  - ambiguous_input
  - judge_disagreement
  - infrastructure_failure
```

23.2 Do not mix infrastructure failures with model failures

Track separately:

• provider timeout;
• evaluator timeout;
• invalid test fixture;
• tool sandbox failure;
• application exception;
• model-quality failure.

Counting a network error as an incorrect answer distorts model comparisons. It is still a reliability failure and should appear in a different metric.

23.3 Prioritize by impact and frequency

```
priority = severity * frequency * confidence_in_diagnosis
```

Fix critical safety failures first. Then target common, high-confidence clusters instead of manually tweaking prompts for isolated examples.

23.4 Map failures to system changes

Failure cluster	Likely intervention
Similar tools confused	Clarify descriptions, remove overlap, add contrastive examples
Required evidence not retrieved	Improve parsing, query, filters, hybrid search, reranker
Evidence retrieved but answer unsupported	Grounding prompt, claim verification, better model
JSON invalid	Constrained decoding, schema simplification
Correct tool, wrong arguments	Typed schema, normalization, argument validation
Agent loops	Explicit state machine, budgets, termination conditions
Judge disagrees with experts	Rewrite rubric, add examples, change or calibrate judge

23.5 Add every confirmed fix to regression coverage

The durable output of debugging is not only the code fix. It is:

- a minimized failing example;
- a named failure category;
- an evaluator that catches it;
- a regression threshold;
- an owner.

24. Common Anti-Patterns

24.1 One overall quality score

It hides whether failures come from retrieval, generation, tools, or operations.

24.2 Evaluating only happy paths

The score looks good because the dataset does not include ambiguity, missing information, similar tools, permission failures, or unanswerable questions.

24.3 Exact matching natural language

It punishes valid paraphrases and encourages output to imitate one reference.

24.4 Using an LLM judge for schema or policy checks

The judge can miss a failure that a parser or authorization engine can prove.

24.5 Trusting a judge without calibration

Fluent explanations from a judge do not prove agreement with domain experts.

24.6 Changing dataset and system simultaneously

The new score cannot be compared with the old score. Re-run the baseline on the new dataset or keep a stable benchmark version.

24.7 Reporting percentages without denominators

An improvement on 20 examples is not equivalent to one on 2,000 examples.

24.8 Optimizing only averages

Averages hide language, tenant, intent, and security failures.

24.9 Letting the model grade its own hidden reasoning

Evaluate observable outputs, evidence, actions, and state transitions. A generated explanation of reasoning is not proof that the underlying process was correct.

24.10 Retrying until the eval passes

Unbounded retries inflate cost and hide instability. Define retry policy before the experiment and report first-pass and eventual-pass rates separately.

24.11 Treating synthetic examples as production distribution

Synthetic data expands coverage but does not estimate real-world frequency unless it was carefully weighted and validated.

24.12 Ignoring evaluator cost

A judge suite can cost more than the application being evaluated. Use code checks first, sample expensive metrics intelligently, and cache immutable results.

25. A Practical Rollout Plan

Week 1: define and instrument

• select one high-value workflow;
• define success and critical failures;
• create a metric tree;
• capture complete traces;

• build 20 manually reviewed examples;
• implement schema, policy, latency, and cost checks.

Week 2: build the first dataset

• expand to 50-100 common examples;
• add known incidents and edge cases;
• label tool, retrieval, and output expectations separately;
• define metadata slices;
• version the dataset;
• run the current production system as baseline.

Week 3: add semantic evaluators

• define narrow rubrics;
• label a human calibration sample;
• implement the judge with structured output;
• measure judge-human agreement;
• adjust thresholds and uncertainty handling;
• add pairwise candidate comparison.

Week 4: automate

• create smoke, regression, and full suites;
• add pull-request and nightly jobs;
• publish per-example artifacts;
• create absolute and regression release gates;
• establish production sampling;
• define the failure-to-regression workflow.

After the first month

• grow the regression set from real failures;
• audit underperforming slices;
• monitor judge drift;
• rotate holdout sets;
• add fault injection and security tests;
• measure business outcomes alongside technical metrics;
• remove metrics that do not lead to decisions.

26. Production Checklist

Product contract

- [ ] Primary task success is defined.
- [ ] Critical failures are listed separately.
- [ ] Every metric maps to a product requirement.
- [ ] Every metric has an owner and action.

Dataset

- [ ] Examples have stable IDs.
- [ ] Dataset versions are immutable.
- [ ] Common, edge, confusion, adversarial, and regression cases exist.
- [ ] Metadata supports important slices.
- [ ] References are reviewed and versioned.
- [ ] Production data is redacted and governed.
- [ ] Holdout data is protected from repeated tuning.

Evaluators

- [ ] Deterministic checks are used before model judges.
- [ ] Rubrics measure one quality dimension at a time.
- [ ] Judge outputs use a strict schema.
- [ ] Judges are calibrated against humans.
- [ ] Judge and rubric versions are recorded.
- [ ] Critical checks do not depend only on a subjective judge.

RAG

- [ ] Ingestion completeness is tested.
- [ ] Retrieval and generation are scored separately.
- [ ] Precision@k, recall@k, and ranking are measured.
- [ ] Faithfulness and correctness are distinct.
- [ ] Citations are checked at claim level.
- [ ] No-answer and permission-filter cases are included.

Agents and MCP

- [ ] Tool selection and execution are separate metrics.
- [ ] Argument schemas and semantic values are checked.
- [ ] Similar tools have confusion tests.
- [ ] Authorization is enforced and evaluated outside the model.
- [ ] Side effects use idempotency and confirmation rules.

• [ ] Error recovery and loops are tested.
• [ ] Complete trajectories are stored.

Operations

• [ ] Model, prompt, index, tool, and evaluator versions are frozen.
• [ ] Latency percentiles and cost per success are tracked.
• [ ] Infrastructure errors are separated from quality failures.
• [ ] CI has fast and full evaluation tiers.
• [ ] Release gates check absolute quality and regression.
• [ ] Production monitoring feeds confirmed failures back into the dataset.

27. Tools and References

The architecture in this guide is framework-independent. A small internal runner is enough to start. Adopt a platform when experiment tracking, annotation, distributed execution, and production tracing justify it.

Examples of available tooling:

• OpenAI evaluation best practices
• LangSmith evaluation concepts
• Ragas metrics
• Arize Phoenix evaluations
• DeepEval metrics
• Promptfoo
• JSON Schema
• OpenTelemetry

Tool choice does not replace evaluation design. The durable assets are the product contract, representative dataset, calibrated evaluators, versioned experiments, and failure-to-regression feedback loop.

Final Principle

The goal is not to prove that an AI system is intelligent. The goal is to know, with evidence, where it works, where it fails, whether a change improved it, and whether it is safe to release.

Start with a small, human-reviewed dataset. Measure components independently. Turn every important production failure into a permanent regression case. Over time, that loop is what converts an impressive demo into a reliable product.