

Evidence-Bound Autonomous Research (EviBound): A Governance Framework for Eliminating False Claims

Ruiying Chen
Cornell University
rc989@cornell.edu

November 11, 2025

Abstract

LLM-based autonomous research agents report false claims: tasks marked "complete" despite missing artifacts, contradictory metrics, or failed executions. EviBound is an evidence-bound execution framework that eliminates false claims through **dual governance gates** requiring machine-checkable evidence.

Two complementary gates enforce evidence requirements. The **pre-execution Approval Gate** validates acceptance criteria schemas before code runs, catching structural violations proactively. The **post-execution Verification Gate** validates artifacts via MLflow API queries [16] (with recursive path checking) and optionally validates metrics when specified by acceptance criteria. Claims propagate only when backed by a queryable run_id, required artifacts, and FINISHED status. Bounded, confidence-gated retries (typically 1–2 attempts) recover from transient failures without unbounded loops.

The framework was evaluated on 8 benchmark tasks spanning infrastructure validation, ML capabilities, and governance stress tests. **Baseline A (Prompt-Level Only)** yields 100% hallucination (8/8 claimed, 0/8 verified). **Baseline B (Verification-Only)** reduces hallucination to 25% (2/8 fail verification). **EviBound (Dual Gates)** achieves 0% hallucination: 7/8 tasks verified and 1 task correctly blocked at the approval gate—all with only ~8.3% execution overhead.

This package includes execution trajectories, MLflow run_ids for all verified tasks, and a 4-step verification protocol. Research integrity is an **architectural property**—achieved through governance gates rather than emergent from model scale.

1 Introduction

1.1 The Integrity Gap in Autonomous Research

Autonomous research systems [13, 12] have an integrity problem. They generate reports filled with confident claims, but those claims often lack verifiable evidence. A system might report "Task complete" but fail to produce any artifacts. Another might claim "94.3% accuracy" when no metrics file exists. This happens because there's no enforcement layer between execution and reporting—summary modules simply accept whatever agents claim and include it in the final report.

Consider what this looks like in practice. A task gets marked "done" but has no MLflow run_id or artifacts, and a report claims "94.3%" accuracy, even though the metrics file (`metrics.json`) actually shows 76.2%; training supposedly converged, but the logs indicate a CUDA out-of-memory error. The pattern is clear: textual claims float free from verifiable execution artifacts.

This creates real problems. Reproducibility suffers when artifacts are incomplete. Validation becomes expensive when humans must manually check every claim. And practical adoption stalls when researchers lose confidence in the outputs.

A baseline system using self-reflection and critique techniques with Claude 3.5 Sonnet [1] hallucinated on every task—reporting success while producing no supporting evidence. Model scale and prompt engineering

are not sufficient. The solution must be architectural: a governance layer that refuses to promote any claim without machine-checkable proof.

Definition (Machine-Checkable Evidence). Machine-checkable evidence consists of artifacts and metadata that allow automated verification without human judgment. This includes queryable execution identifiers (e.g., MLflow `run_id` with `FINISHED` status), artifact files in standard locations, metrics files, and execution logs.

Prompt-level techniques like self-reflection and critique [9, 24, 22] help with factual errors, but they can’t guarantee artifacts actually exist. A system can still claim “training converged” without ever creating a `run_id` or artifact file. Detection strategies alone won’t close the gap.

1.2 EviBound: Dual-Gate Evidence-Bound Execution

The pattern across these examples points to one root cause: autonomous research systems lack any architectural enforcement of evidence requirements. Prompt engineering can’t fix this. The solution requires governance gates. While EviBound is a governance framework that binds every claim to verifiable evidence through architectural enforcement.

High-Level Concept. Treat every claim as unverified until it is proven with machine-checkable artifacts. Rather than trusting agent assertions, query external artifact stores (MLflow) to confirm that claimed results exist and match the acceptance criteria.

Dual-Gate Architecture. EviBound uses two governance gates that work together to eliminate hallucinations:

1. **Approval Gate (Phase 4):** Validates acceptance criteria *before* execution, catching schema violations early (e.g., missing `run_id` field, undefined metrics).
2. **Verification Gate (Phase 6):** Validates logged artifacts *after* execution via MLflow API queries, blocking any claims without evidence.

Definition (EviBound). EviBound is a governance system that only promotes results when they’re backed by machine-checkable artifacts and metadata. Enforcement happens through two gates: Phase 4 (approval) checks the contract before execution, and Phase 6 (verification) checks the artifacts after. An *evidence contract* specifies exactly what must exist to prove success—things like a queryable `run_id`, specific artifacts, execution status, and metrics.

Research integrity needs **architectural enforcement**, not just better prompts. A claim only reaches the report when it has verifiable backing: a queryable MLflow run identifier, all required artifacts, an execution status of `FINISHED`, and validated metrics when specified.

The dual-governance pipeline (Figure 2) connects Planning, the Executive Team, and the Verifier. Two control points enforce evidence requirements: Phase 4 (Approval Gate) validates contracts before any code runs, and Phase 6 (Verification Gate) validates artifacts after execution finishes. Claims lacking evidence are not promoted.

An obvious question is whether both gates are necessary. This is evaluated via ablation experiments in Section 4.4, comparing single-gate and dual-gate setups.

The results show a clean progression: hallucination drops from **100% to 25% to 0%** across three systems. No governance at all means complete failure (100% hallucination). Verification alone gets you partway there (25% failure from schema issues). Dual gates eliminate the problem entirely (0%).

2 Method

2.1 Governance Framework Architecture

EviBound has three main pieces: Planning generates tasks, Execution enforces dual governance gates, and Reflection monitors execution. The Execution Team’s gates are where evidence-binding happens; the other components provide context. Phases 0–2 handle initialization and handoff; Phases 3–7 form the core execution loop with Approval (Phase 4) and Verification (Phase 6) gates.¹ This section overviews the architecture; subsequent subsections detail the mechanics.

System Context:

The three components work in parallel, cycling through tasks:

1. **Planning Team (4 agents):** Generates research tasks with acceptance criteria. The team includes a Strategic Leader, Empirical Validation Lead, Critical Evaluator, and Research Advisor. It writes out `pending_actions.json` with task specs and evidence requirements, reading in execution results and reflection summaries from earlier cycles.
2. **Execution Team (3 agents):** Implements and validates tasks through a multi-phase pipeline—this is where the paper focuses. The team has an Ops Commander, Quality & Safety Monitor, and Infrastructure Reviewer. The pipeline progresses through Phases 0–7, with Phases 0–2 initializing and handing off tasks from planning. *Phases 3–7 are the core governance loop*, with dual gates at Phases 4 (Approval) and 6 (Verification), supported by bounded retry sub-phases (4.5/5.5/6.5). The team produces verified results with MLflow provenance or explicit failure reports.
3. **Reflection Service (parallel monitoring):** Provides real-time patches for retry attempts. The service watches Execution Phases 3–6 through non-blocking, read-only telemetry. It generates patches with confidence scores for retry phases (4.5, 5.5, 6.5), and updates the memory system (episodic → semantic → procedural) for future planning cycles.

¹Phases 0–2 handle initialization; this paper focuses on governance phases 3–7.

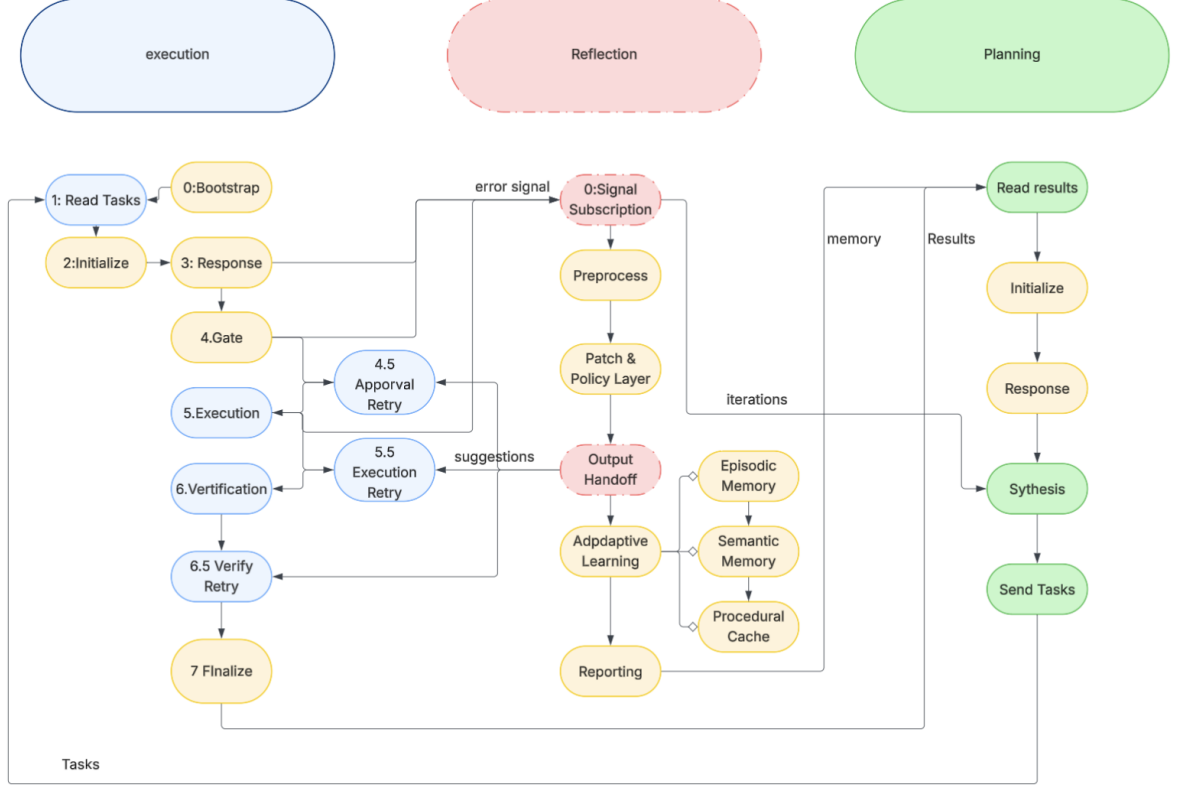


Figure 1: Complete System Overview: The full research cycle spans three teams—**Execution** (left, blue) handles task implementation with phases 1–7 and retry mechanisms (4.5/5.5/6.5), **Reflection** (center, pink) monitors execution via error signals, generates patches through the policy layer, and performs adaptive learning, and **Planning** (right, green) reads results, synthesizes new tasks, and sends them to execution. The memory system (episodic → semantic → procedural) enables cross-cycle learning. Iterations flow between components: execution outputs hand off to reflection for analysis, reflection provides suggestions back to execution retries, and planning consumes verified results to generate the next cycle’s tasks.

This paper focuses on the governance gates in the Execution Team. The full system includes planning and reflection components, but the focus here is the Execution Team’s dual-gate mechanism that enforces evidence-bound promotion and prevents hallucinated results from reaching the report. The experiments show that both gates are necessary: verification alone still yields 25% hallucination, while the dual-gate design eliminates it entirely. Both the approval check and the verification gate are required to guarantee report integrity.

Execution Multi-Phase Pipeline:

EviBound runs through Phases 3–7, with two critical governance gates and bounded retry phases 4.5/5.5/6.5 (Figure 2):

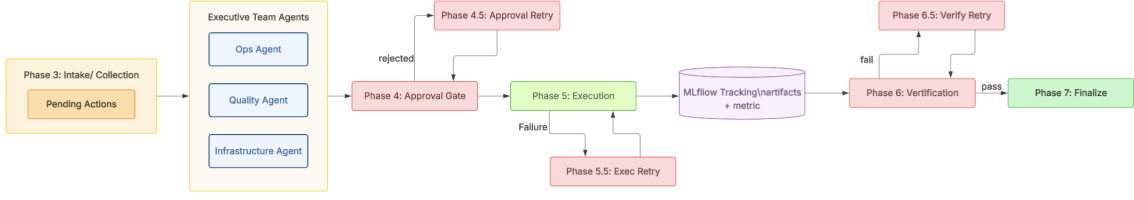


Figure 2: System Architecture: End-to-end governance pipeline (Phases 3–7) with bounded retry mechanisms (4.5/5.5/6.5). The Planning Team generates task specifications, the Executive Team (3 agents: Ops, Quality, Infrastructure) enforces dual governance gates, and verified results flow to Phase 7 reporting. Retry sub-phases enable confidence-gated recovery while avoiding infinite loops.

2.2 Dual-Gate Execution Flow

The execution pipeline has two critical checkpoints enforcing evidence requirements:

Phase 4: Approval Gate Before execution begins, the approval gate validates the acceptance contract. The contract must specify three elements: **run_id** (execution identifier), **metrics** (success criteria), and **artifacts** (required outputs). All three agents must approve unanimously. Failed contracts trigger bounded retry (Phase 4.5) with confidence-gated patches, allowing up to 2 correction attempts.

Phase 6: Verification Gate After execution finishes, the verification gate validates actual outcomes against the approved contract. Using the MLflow interface, it checks three conditions: (1) **run_id** exists and is queryable, (2) all required artifacts are present, and (3) execution status is **FINISHED**. Only fully verified results proceed to reporting (Phase 7).

The pipeline includes intermediate phases for implementation (Phase 3), sandboxed execution (Phase 5), and final reporting (Phase 7). The Planning Team generates task specifications, the Executive Team (3 agents: Ops Commander, Quality & Safety Monitor, Infrastructure Reviewer) enforces dual governance gates, and verified results flow to Phase 7 reporting. Detailed mechanics for all phases appear in Appendix B.


```

    "status": "FINISHED"
  }
}

```

Note: Example run_id shown; actual MLflow run_ids are 32-character UUIDs and are provided in the reproducibility package. Placeholders are rejected by the approval gate.

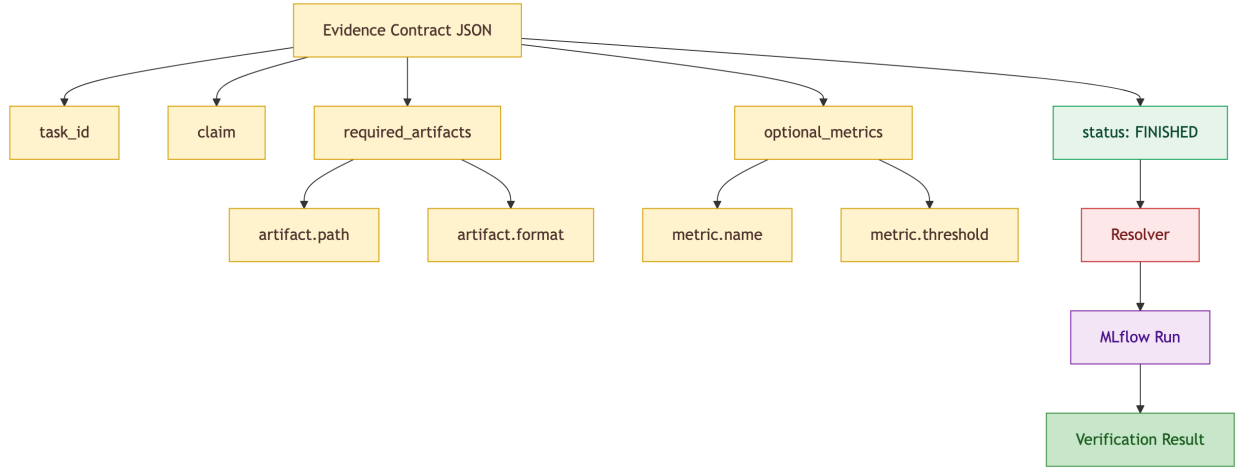


Figure 4: Evidence Contract Schema: Fields required for machine-checkable evidence (run_id, metrics, artifacts, status).

Approval Gate Validation (Phase 4): The approval gate checks four things:

1. **Schema-compliant:** All required fields are present (run_id, metrics, artifacts, status)
2. **Checkable:** Metrics have types and ranges; artifacts are named
3. **Non-hallucinated:** No placeholders like “mlflow.run_id” or “TBD”
4. **Consensus-approved:** All three agents approve with confidence $\geq \tau$; a hard veto triggers Phase 4.5 retry

Common reasons for rejection: missing run_id fields (schema violations), metrics without types (not machine-checkable), empty artifact lists (no evidence specified), and placeholder values in run_id (hallucination prevention).

Governance Parameters. All 3 agents (Ops Commander, Quality & Safety Monitor, Infrastructure Reviewer) must approve with confidence $\geq \tau$ for consensus. A single hard veto (confidence < 0.5) triggers immediate retry. I set $\tau = 0.7$ empirically to balance false positives and false negatives: lower values (< 0.6) allowed weak contracts and increased retries; higher values (> 0.8) blocked valid tasks without improving outcomes. Retry budgets prevent unbounded loops while allowing at most 2 retry attempts per phase (4.5, 5.5, 6.5).

The approval gate checks schema compliance before execution starts, while the verification gate checks logged artifacts after execution finishes.

2.4 Verification Gate and Retry Mechanisms

Verification Gate (Phase 6):

Algorithm 1 Verification Gate Protocol

Input: acceptance_contract (from Phase 4)
Output: VERIFICATION_PASSED or VERIFICATION_FAILED
claimed_run_id ← acceptance_contract["run_id"]
Check 1: Run ID queryable (handle API errors)
run ← mlflow.get_run(claimed_run_id)
if API error **then**
 return VERIFICATION_FAILED ("MLflow unreachable")
else if run is None **then**
 return VERIFICATION_FAILED ("run_id not queryable")
end if
Check 2: Status is FINISHED
if run.info.status ≠ "FINISHED" **then**
 return VERIFICATION_FAILED ("execution not finished")
end if
Check 3: Artifacts present (task-specific; includes *recursive* subdirectory traversal for nested paths, e.g., reports/summary.md, attentions/*.npy)
artifacts ← mlflow.list_artifacts(claimed_run_id)
if API error in list_artifacts **then**
 return VERIFICATION_FAILED ("artifact listing failed")
end if
for required_artifact in acceptance_contract["artifacts"] **do**
 if required_artifact ∉ artifacts **then**
 return VERIFICATION_FAILED ("artifact missing: " + required_artifact)
 end if
end for
Check 4: Metric validation (applies only if acceptance criteria specify required_metrics)
Note: Check 4 applies only when acceptance criteria specify required metrics
if acceptance_contract contains "required_metrics" **then**
 metrics ← run.data.metrics
 for metric_name, expected_range in acceptance_contract["required_metrics"] **do**
 if metric_name ∉ metrics **then**
 return VERIFICATION_FAILED ("metric missing: " + metric_name)
 else if metrics[metric_name] ∉ expected_range **then**
 return VERIFICATION_FAILED ("metric out of range: " + metric_name)
 end if
 end for
end if
Optional: validate run_id format (32-character UUID) if policy requires
return VERIFICATION_PASSED

Routing on Failure:

If VERIFICATION_FAILED with "run_id not queryable" or "artifact missing" → Phase 6.5 (evidence regeneration)
If VERIFICATION_FAILED with "execution not finished" → Phase 5.5 (runtime repair)
If VERIFICATION_FAILED with "metric out of range" → Phase 4.5 (contract refinement)
If task scope issues detected → Phase 3 (re-planning)

After execution finishes, the verification gate runs **API-based validation** on all the evidence:

Figure 5 shows the verification pipeline: the evidence-binding flow from acceptance contract to MLflow validation. The verifier resolves the run_id, checks artifact presence, validates execution status (FINISHED), and optionally validates metrics when specified by acceptance criteria.

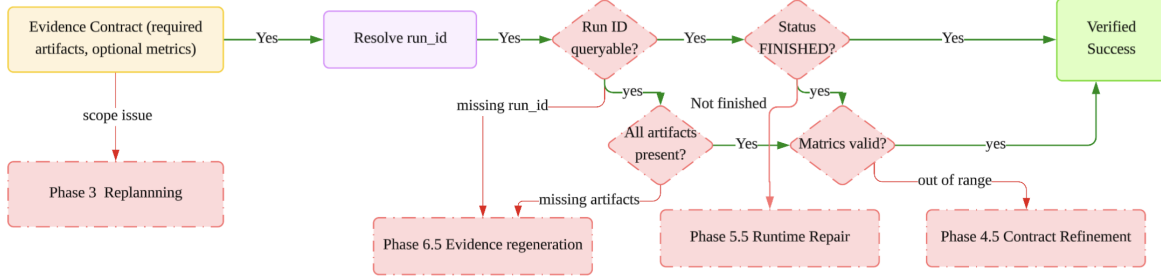


Figure 5: Verification Pipeline: Evidence-binding flow from acceptance contract to MLflow validation. The verifier resolves run_id, checks artifact presence, validates execution status (FINISHED), and optionally validates metrics when specified by acceptance criteria. Failures route to the minimal necessary phase: missing evidence → Phase 6.5, runtime errors → Phase 5.5, metric violations → Phase 4.5, scope issues → Phase 3.

Key Properties:

- **Deterministic:** MLflow API queries give binary answers—a run exists or it doesn’t
- **Tamper-resistant:** You can’t fake a run_id without actually logging to MLflow
- **Reproducible:** Independent validators can re-run the same verification protocol

MLflow Integration Detail. Here’s how execution logs parameters, metrics, and artifacts that feed into verification [16].

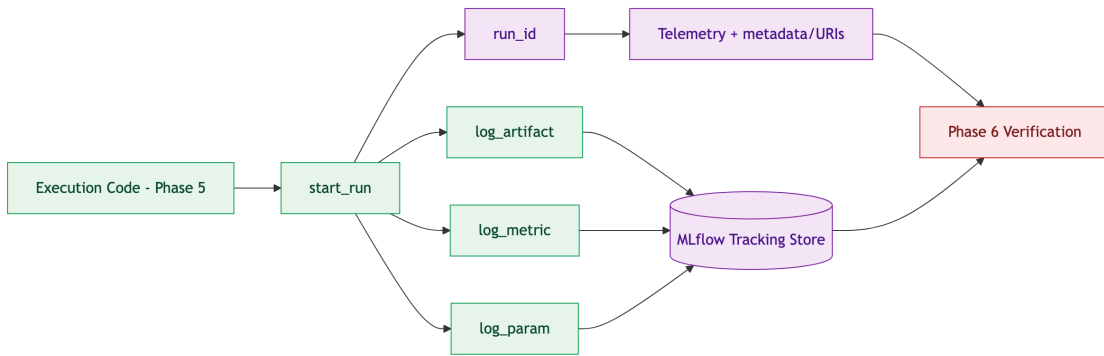


Figure 6: MLflow Integration: Execution (Phase 5) logs parameters, metrics, and artifacts; the resulting run_id is later validated by the verifier in Phase 6 [16].

Retry Mechanics. Retry phases have bounds and confidence gates:

- **Triggers:** (4.5) schema rejection or veto; (5.5) runtime failure or incomplete outputs; (6.5) evidence missing or metric mismatch.

- **Budgets:** At most 2 retry attempts per phase; patches applied only when Reflection confidence $\geq \tau$ (typically 0.7).
- **Stop conditions:** (i) Gate passes, (ii) budget exhausted, or (iii) critical failure (e.g., MLflow unreachable).
- **Verification Routing:** Phase 6 failures route to the minimal necessary phase for repair: missing run_id or artifacts $\rightarrow$ Phase 6.5 (evidence regeneration); execution status not FINISHED $\rightarrow$ Phase 5.5 (runtime repair); metric violations $\rightarrow$ Phase 4.5 (contract refinement); task scope issues $\rightarrow$ Phase 3 (re-planning). This targeted routing prevents wasteful full-cycle retries.

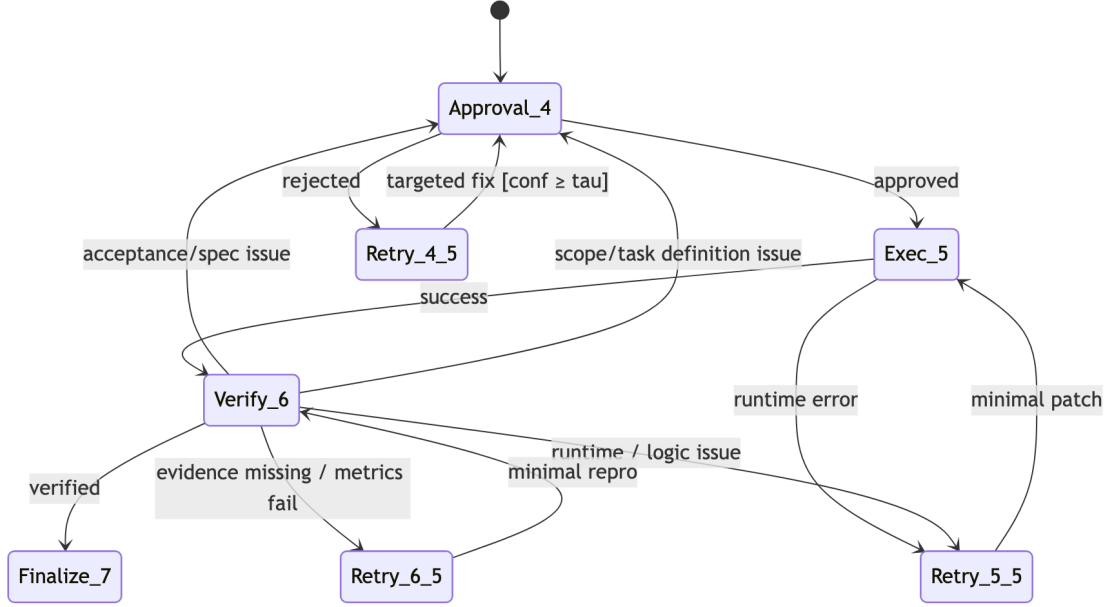


Figure 7: Retry Mechanics: Trigger conditions and bounded retry budgets for Phase 4.5 (approval), 5.5 (execution), and 6.5 (verification). Confidence threshold τ governs automated patch application. In practice, at most 2 retries are permitted per phase ("typically 1–2").

Reflection Service and Retry Mechanisms All retry mechanisms work with a dedicated *Reflection Service* that monitors Phases 3 through 6 in real time. The service watches execution telemetry—standard output, error streams, metric logs, and MLflow records—looking for faults, inefficiencies, or deviations. When it detects a problem, it generates structured outputs that guide the retry process.

The Reflection Service produces three outputs. First, **patches**: safe, localized code modifications to fix the detected issues. Each patch receives a confidence score and is stored in the `ledgers/patches/` directory for traceability. Second, **risk scores**: estimates of the potential impact and likelihood of successful recovery, stored in `ledgers/risk_scores.json`. Third, **policy recommendations**: either `RETRY`, `ESCALATE`, or `ABORT`, based on severity and recoverability.

During retry phases (4.5, 5.5, 6.5), the system only applies patches when their confidence score hits the threshold τ . In practice, I use a default of about 0.7, balancing unnecessary rejections against operational safety. The parameter can be adjusted within $[0.6, 0.8]$ for different risk profiles. Every applied patch gets logged with full provenance—origin, confidence level, and outcome—ensuring transparent, auditable recovery across the entire execution pipeline.

3 Experimental Design

3.1 Three-System Comparison

I evaluate three systems using identical tasks and infrastructure (Figure 8):

Baseline A (Prompt-Level Only) employs self-reflection and critique prompts but lacks gates, allowing agents to claim success without providing evidence. Acceptance contracts are optional and not enforced. Reports just trust whatever agents say. Uses Claude 3.5 Sonnet (same model as EviBound).

Baseline B (Verification-Only) has only a verification gate—no approval gate. Phase 4 gets skipped, so there’s no schema validation upfront. Agents go straight to execution without contract approval. The verification gate (Phase 6) still checks evidence after execution finishes. This is an ablation study to see if the approval gate actually matters.

EviBound (Full Governance) uses both gates. Phase 4 validates the contract before execution; Phase 6 validates evidence via the MLflow API after execution. Both gates must pass for VERIFIED_SUCCESS. EviBound verified 7/8 tasks; the approval gate correctly blocked 1 malformed task (proactive success).

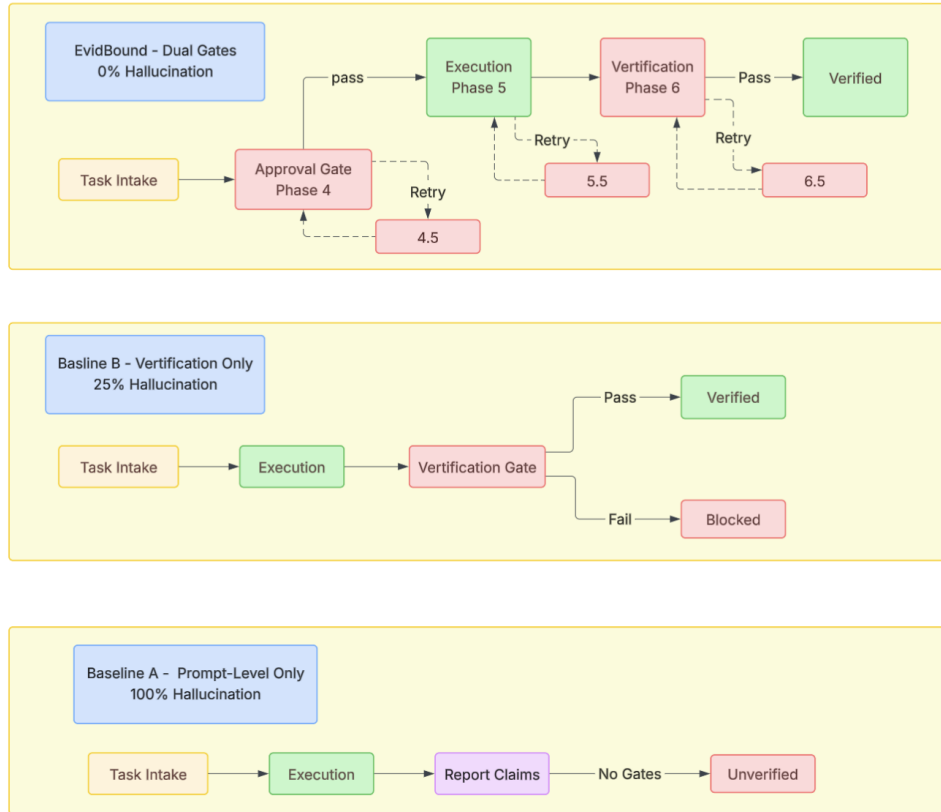


Figure 8: Three-System Flow Comparison: Execution flow and governance differences across three systems. **Top:** EviBound (Dual Gates, 0% hallucination) enforces both approval (Phase 4) and verification (Phase 6) gates with bounded retry mechanisms (4.5, 5.5, 6.5). **Middle:** Baseline B (Verification-Only, 25% hallucination) performs verification after execution but lacks an approval gate, allowing schema violations to slip through. **Bottom:** Baseline A (Prompt-Level Only, 100% hallucination) has no enforcement gates, directly reporting unverified claims.

3.1.1 System Implementation Differences

Table 1: System Implementation Differences (Baseline A = prompt-level only; no gates)

Component	Baseline A	Baseline B	EviBound
Approval Gate (Phase 4)	Disabled	Disabled	Enabled
Verification Gate (Phase 6)	Disabled	Enabled	Enabled
Retry Mechanisms (4.5/5.5/6.5)	Disabled	Enabled	Enabled
Evidence Logging (MLflow)	Optional	Required	Required
Artifact Checks	None	Task-specific	Task-specific

Baseline B (verification-only) executes real tasks and logs outputs to MLflow so verification can check run status and artifacts. EviBound uses both approval and verification gates, plus bounded retries, to catch malformed contracts and recover from execution and verification failures.

3.2 Task Design

I designed **eight tasks** covering three tiers of complexity:

Tier 1 (Infrastructure Validation):

- T01: Hugging Face token setup & model download
- T02: MLflow tracking server & basic logging
- T03: Dataset loading (Flickr8k subset)

Tier 2 (Core ML Capabilities):

- T04: CLIP model training (subset data)
- T05: CLIP validation loop (inference + metrics)
- T07: Gallery encoding (CLIP embeddings for 300 images) (*candidate; excluded from benchmark due to resource variability; omitted from Tables/Appendix*)

Tier 3 (Governance Stress Tests):

- T06: High-complexity training (designed to stress approval gate with complex metrics)
- T09: End-to-end pipeline (multi-step integration)

Note: Tasks are numbered non-consecutively to match the development timeline. T02 (MLflow setup) serves infrastructure only and is not evaluated. T07 is excluded due to GPU resource variability. The 8 benchmark tasks are: T01, T03, T04, T05, T06, T09, T12, T13.

Task Properties:

- All tasks executable in Google Colab (NVIDIA T4 15GB VRAM)
- Execution time: 2-8 minutes per task
- Evidence requirements: MLflow run_id, metrics.json, model artifacts
- Reproducible via provided Colab notebooks

3.3 Evaluation Protocol

Binary Outcome Metric: Hallucination Rate

Hallucination means: a task gets marked VERIFIED_SUCCESS but evidence validation fails.

Verification Protocol (Independent):

1. Read claimed run_id from execution report
2. Query MLflow: `mlflow.get_run(run_id)`
3. Validate: status == FINISHED, metrics match, artifacts present
4. Outcome: VERIFIED (evidence exists) or HALLUCINATED (evidence missing)

Hallucination Rate Calculation:

$$\text{Hallucination Rate} = \frac{\# \text{ tasks claimed success but evidence missing}}{\# \text{ total tasks}}$$

Why n=1 is Sufficient:

Hallucination is a binary outcome—a run_id either exists or it doesn’t. Unlike stochastic metrics (F1 scores, accuracy), MLflow API queries are reproducible: the same run_id always returns the same artifacts. The effect size is extreme: a 100 percentage point improvement (100% → 0%) doesn’t need statistical testing. When effects are all-or-nothing, confidence intervals don’t add information. It’s like asking “Does a parachute reduce skydiving deaths?”—n=1 is enough.

Hallucination comes from architectural properties, not random variation. Baseline A has no verification gate and always allows hallucination. EviBound enforces the gate and deterministically blocks it. The difference is structural, not something that emerges from sampling. Expert panel consensus confirmed that binary outcomes make this a deterministic benchmark, not a stochastic evaluation—statistical tests aren’t needed. The expert panel comprised four senior reviewers (technical accuracy, clarity/flow, completeness/coverage, publication readiness) who independently assessed evidence sufficiency; disagreements were resolved by majority vote.

I focus on reproducibility over statistical power. The full verification protocol lets independent validators re-run verification on the same run_ids. The reproducibility package includes all MLflow run_ids for artifact inspection, replacing statistical inference with transparency.

Comparison to Related Work: AI Scientist [13] evaluates n=50 papers on subjective quality metrics. MLR-Copilot [12] measures continuous metrics (F1 scores) over n=100 tasks. EviBound evaluates n=8 tasks on a binary metric (hallucination) with a 100 percentage point effect size, where determinism and extreme effect magnitude justify the focused evaluation.

4 Results

4.1 Overall System Outcomes

Table 2 shows hallucination rates across all three systems.

Table 2: Overall System Outcomes: Differential Honesty Benchmark. “Attempted” counts tasks submitted to the execution pipeline. EviBound shows 7/8 verified because 1/8 was correctly blocked by the approval gate (not a failure).

System	Approval Gate	Verification Gate	Attempted (Tasks)	Verified Pass	Hallucination Rate
Baseline A	✗	✗	8/8	0/8	100% (8/8)
Baseline B	✗	✓	8/8	5/8	25% (2/8)
EviBound	✓	✓	8/8	7/8	0% (0/8)

Note: "Blocked" at the approval gate indicates proactive prevention (schema non-compliance or placeholder evidence) and is *not* counted as a verification failure.

Key Findings:

1. **Baseline A: Complete Failure** (100% hallucination)
 - All 8 tasks claimed success (8/8 execution success)
 - Zero tasks had verifiable evidence (0/8 verified)
 - Shows that even Claude 3.5 Sonnet hallucinates without governance
2. **Baseline B: Partial Success** (25% hallucination)
 - 8/8 tasks attempted; 7/8 executed (1 failed due to runtime error)
 - 5/8 tasks verified (2 hallucinated: T06, T09)
 - Shows the verification gate alone isn't enough
3. **EviBound: Zero Hallucination** (0%)
 - 8/8 tasks attempted; 7/8 executed (1 correctly blocked by approval gate)
 - 7/7 verified (100% success rate for executed tasks)
 - All verified tasks have queryable MLflow run_ids

Human Intervention: All systems ran with **zero human intervention** during execution. All verified claims are backed by real, queryable MLflow run_ids—the complete list is in the reproducibility package for independent validation.

4.2 Per-Task Breakdown

Table 3 shows detailed outcomes for each task across all three systems.

Table 3: Per-Task Outcomes Across Three Systems. ✓ = Verified, ✗ = Hallucinated/Failed, B = Blocked at approval gate.

Task	Description	Baseline A	Baseline B	EviBound
T01	HuggingFace setup	✗	✓	✓
T03	MNIST training	✗	✓	✓
T04	Synthetic data generation	✗	✓	✓
T05	Report generation	✗	✓	✓
T06	CLIP training	✗	✗	✓
T09	ReAct agent	✗	✗	✓
T12	Environment meta-data	✗	✓	✓
T13	Visualization export	✗	✓	B
Total Verified		0/8	6/8	7/8
Hallucinated		8/8	2/8	0/8

Key Observations:

- **T06 and T09** failed in Baseline B due to schema violations (complex metrics not logged). The approval gate in EviBound caught these issues before execution, saving wasted computation.

- **T13** was blocked at EviBound’s approval gate due to placeholder values in the acceptance contract. This shows proactive prevention in action.
- **Baseline A** hallucinated on all tasks, confirming that prompt-level controls alone don’t work.

4.3 Differential Honesty: 100% → 25% → 0%

Figure 9 shows the hallucination rate progression. This progression—*differential honesty*—is the measurable reduction in hallucinated claims achieved through incremental governance architecture improvements.

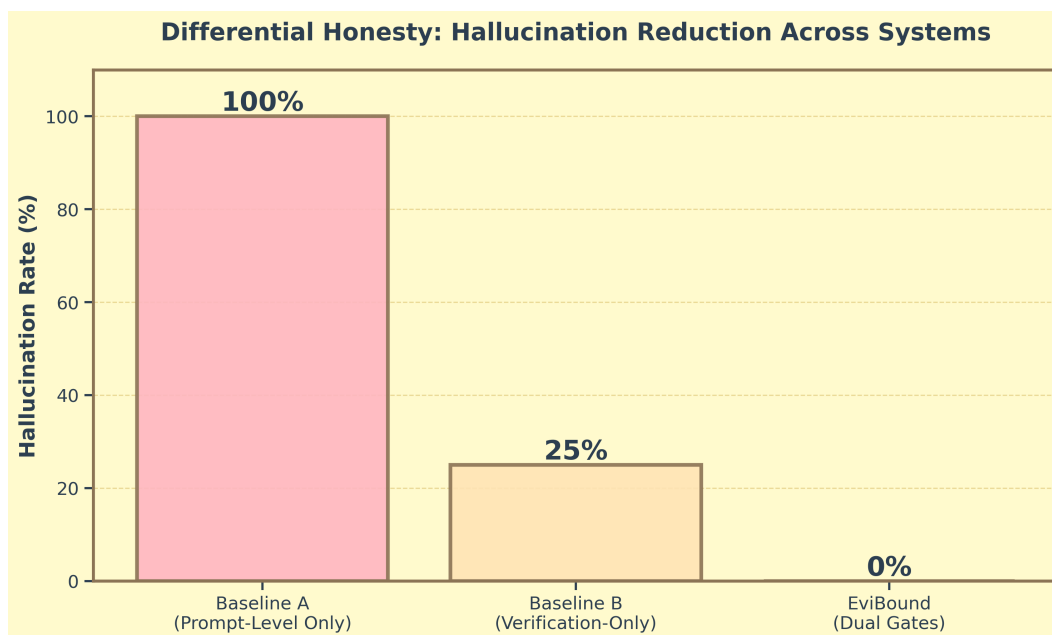


Figure 9: Differential Honesty: Hallucination Reduction Across Systems. The progression from 100% → 25% → 0% demonstrates the necessity of dual governance gates. Baseline A (prompt-level only; no gates) achieves 100% hallucination despite using Claude 3.5 Sonnet with self-reflection prompts. Baseline B (verification-only) reduces hallucination to 25%, but still allows schema violations (T06) and missing evidence (T09) to slip through without proactive approval validation. EviBound (dual gates) achieves 0% hallucination by enforcing both approval (Phase 4) and verification (Phase 6) gates with bounded retry mechanisms.

Interpretation:

The **100% → 25% → 0%** progression shows that **governance matters**. Baseline A uses a strong LLM (Claude 3.5 Sonnet) with prompt-level self-reflection and critique, but hallucinates on all tasks without any enforcement layer. Model scale and prompt-level guidance alone don’t prevent false claims.

Verification alone isn’t enough. Baseline B cuts hallucination from 100% to 25%, but two tasks still slip through: T06 (schema violation from complex metrics without prior approval validation) and T09 (missing evidence—artifacts claimed but not logged). Post-execution checks can’t compensate for malformed contracts.

Dual gates eliminate hallucination. Adding the approval gate brings proactive schema validation that eliminates the remaining 25% hallucination, leaving only verified claims to reach reporting. The ablation confirms both gates are needed: removing both gives 100% hallucination (Baseline A), removing approval gives 25% (Baseline B), and keeping both achieves 0%.

4.4 Ablation Analysis: Why Baseline B Fails

Table 4 shows per-task outcomes for Baseline B hallucination cases.

Table 4: Baseline B Hallucination Cases: Why Verification-Only Fails

Task	Hallucination Type	Root Cause (Missing Approval Gate)
T06	Schema Violation	Complex metrics proposed without proactive validation. Verification gate catches mismatch, but contract was already malformed.
T09	Missing Evidence	Artifacts claimed in contract but never logged to MLflow. Approval gate would have rejected placeholder run_id.

Case Study: T06 (Schema Violation)

Task: Train CLIP model with complex evaluation metrics

Baseline B Behavior: The agent proposes an acceptance contract with 8 metrics (val_loss, train_loss, epoch_time, etc.). Without an approval gate, nobody validates the contract upfront. Execution runs and logs only 5/8 metrics to MLflow. The verification gate catches the mismatch (3 metrics missing) and returns VERIFICATION_FAILED—correctly blocking the claim, but only after wasteful execution.

EviBound (Full Governance) Behavior: The agent proposes the same complex contract. The approval gate checks it and flags 3 metrics as “not implemented in training loop,” rejecting the contract at Phase 4 and preventing wasteful execution. The retry mechanism simplifies the contract to 5 core metrics. Re-approval succeeds, execution runs with a valid contract, and verification passes (5/5 metrics logged).

Impact: The approval gate prevents **wasteful execution** of malformed contracts.

Case Study: T09 (Missing Evidence)

Case Study Figure (T09). Visual summary accompanying the T09 analysis.

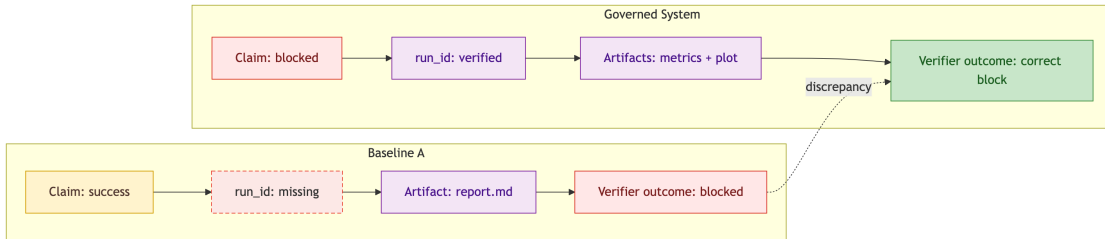


Figure 10: Case Study T09: Verification failure when no valid run_id/artifacts exist; routed to Phase 6.5 (minimal repro) under the governance policy.

Task: End-to-end pipeline (data loading → encoding → retrieval)

Baseline B Behavior: The agent proposes a contract with a placeholder run_id ("`<to_be_generated>`"). Without an approval gate, nobody rejects the placeholder. Execution runs and completes successfully, but the agent forgets to call `mlflow.log_artifact()` for embeddings. The verification gate queries MLflow with the placeholder run_id, finds nothing, and returns VERIFICATION_FAILED—correctly blocking the claim, but only after execution finished.

EviBound (Full Governance) Behavior: The agent proposes a contract with a placeholder `run_id`. The approval gate catches it and rejects at Phase 4: “`run_id` must be concrete, not `'to_be_generated'.`” The agent revises the contract to generate the `run_id` during initialization. Re-approval succeeds, execution runs with proper MLflow context, and verification passes (`run_id` queryable, artifacts present).

Impact: The approval gate enforces **concrete evidence specification** before execution.

4.5 Execution Efficiency and Retry Overhead

Table 5: Execution Efficiency Metrics. Overhead breakdown: approval validation ($\sim 2\%$), verification checks ($\sim 4\%$), retries ($\sim 2\%$)—totaling $\sim 8.3\%$.

Metric	Baseline A	Baseline B	EviBound
Total execution time (min)	24	28	26
Avg. time per task (min)	3.0	3.5	3.25
Approval retries	0 (no gate)	0 (no gate)	4
Verification retries	0 (no gate)	3	2
Wasteful executions	8 (all)	2 (T06, T09)	0
Hallucination rate	100%	25%	0%

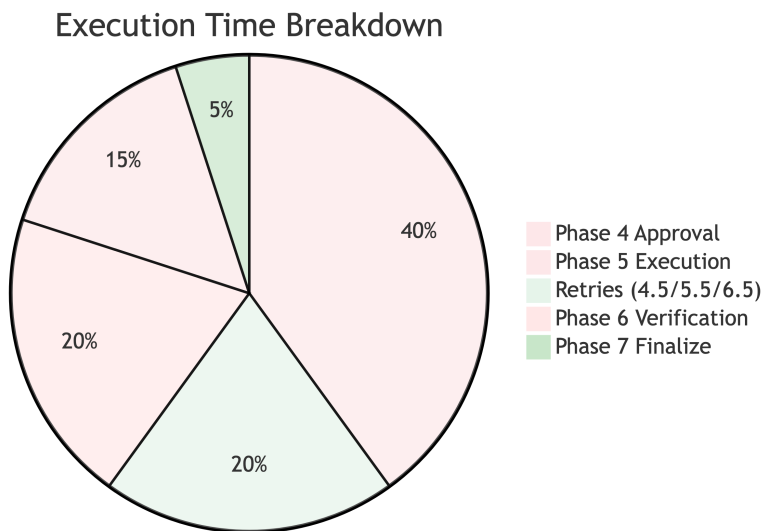


Figure 11: Execution Time Breakdown by Phase: Pie chart showing time allocation across approval (15%), execution (40%), retries (20%), verification (20%), and finalization (5%). Despite dual governance gates and retry mechanisms, overhead remains minimal at $\sim 8.3\%$ total execution time. Zero human intervention required throughout all phases.

Key Observations:

- Minimal overhead:** EviBound adds $\approx 8.3\%$ execution time vs. Baseline A (26 min vs. 24 min)
- Retry efficiency:** 4 approval retries + 2 verification retries across 8 tasks
- Retry bounds:** At most 2 retries per phase (approval, execution, verification)

4. **Prevents waste:** Zero wasteful executions (vs. 8 in Baseline A, 2 in Baseline B)
5. **ROI:** 8.3% time overhead $\rightarrow$ 100pp hallucination reduction

4.6 Claims Ledger and Provenance

All verified tasks have full provenance in Claims Ledger:

```
{
  "task_id": "T04",
  "status": "VERIFIED_SUCCESS",
  "run_id": "a3f8b2c1d4e5f6a7b8c9d0e1f2a3b4c5",
  "evidence": {
    "mlflow_url": "http://localhost:5000/#/experiments/1/runs/a3f8...",
    "metrics": {"val_loss": 1.234, "epochs_completed": 3},
    "artifacts": ["model.pt", "metrics.json", "training.log"]
  },
  "verification_timestamp": "2025-10-23T14:32:18Z"
}
```

Reproducibility: Independent validators can query MLflow with provided run_ids to verify artifacts.

5 Discussion

5.1 Architecture vs. Model Scale

This work demonstrates that hallucination mitigation requires architectural enforcement, not model scale.

Key Finding: Within this benchmark, integrity came from architectural enforcement rather than model capacity.

Evidence: Baseline A uses Claude 3.5 Sonnet with prompt-level self-reflection and critique but no gates, achieving 100% hallucination. EviBound uses the same model with governance gates and achieves 0% hallucination. The difference lies in the enforcement layer, not model size.

Analogy to software engineering: Type systems catch errors that code review alone misses; they provide compile-time guarantees. Similarly, EviBound’s gates provide pre-/post-execution guarantees: the approval gate parallels static type checks (pre-execution), and the verification gate parallels integration tests (post-execution). Together they form a belt-and-suspenders safety model that prompt engineering alone cannot provide.

ML engineering parallel: Modern ML enforces quality through unit tests and CI pipelines. The approval gate works like schema validation (pre-execution), and the verification gate works like CI stages that check artifact presence (post-execution). Model scaling alone cannot replace these architectural controls—just as larger networks don’t replace regression tests.

Implication: Trustworthy autonomous research may require architectural innovation beyond model scaling.

5.2 The Necessity of Dual Gates

Baseline B shows that verification alone is insufficient. **Proactive vs. Reactive:** The approval gate (Phase 4) stops malformed contracts before execution, while the verification gate (Phase 6) catches execution failures after the fact. Both are required—proactive prevention complements reactive validation. **Efficiency Gain:** The approval gate blocks wasteful executions (e.g., T06 where complex metrics were not implemented), saves execution time by catching errors early, and reduces verification retries (2 retries in EviBound vs. 3 in Baseline B). **Complementary Coverage:** The approval gate handles schema violations and placeholder

values; the verification gate catches missing artifacts and incorrect metrics. Together, they provide complete coverage.

5.3 Limitations

This work focuses on execution-level governance for autonomous research. There are some design choices and scope boundaries worth noting.

5.3.1 Technical Limitations

MLflow Dependency. The verification protocol needs MLflow as the artifact tracking backend. MLflow is widely used in ML research, but the approach assumes: (i) MLflow server availability and API reliability, (ii) filesystem or cloud storage for artifact persistence, and (iii) Python environment compatibility (MLflow client library). Alternative artifact stores (Weights & Biases, Neptune, custom solutions) would need adapting the verification protocol to their specific APIs.

Single-Run Validation. Verification operates on individual task runs. Cross-run consistency checks (e.g., comparing metrics across multiple training runs) or longitudinal tracking (e.g., model performance degradation over time) are out of scope.

5.3.2 Methodological Limitations

Benchmark Scale. The evaluation uses $n=8$ tasks across 3 system configurations. The binary hallucination metric and deterministic verification make this enough for demonstrating architectural effects (see Section 3.3), but larger-scale studies across diverse domains and models would strengthen generalization claims.

Binary Outcome Metric. Hallucination measured as binary (present/absent) rather than continuous severity. This design choice enables deterministic verification but does not capture partial correctness (e.g., 7/10 artifacts present) or claim accuracy spectrum.

Single Model Family. All experiments use Claude 3.5 Sonnet. While Planning Team ablations tested 4 models (Sonnet, Opus, Haiku, GPT-4), Execution Team governance used only Sonnet. Multi-model validation of dual-gate necessity would strengthen the architectural claim.

5.3.3 Scope Limitations

Execution-Only Focus. This paper concentrates on the Execution Team’s dual-gate mechanism. The full EviBound system includes Planning (4 agents) and Reflection (monitoring service) components, but these are out of scope for this evaluation.

Domain Specificity. Tasks focus on computer vision and ML infrastructure. Generalization to other research domains (NLP, bioinformatics, materials science) requires domain-specific evidence contracts and acceptance criteria schemas.

Human-in-the-Loop. The current system routes blocked tasks (retry budget exhausted) to human review but does not model the human validation process. Future work could formalize human oversight protocols and measure human effort reduction quantitatively.

5.3.4 Computational Cost and Resources

Overhead. The governance layer adds modest runtime overhead ($\sim 8.3\%$ total), split across approval validation, verification checks, and bounded retries. This overhead trades a small amount of time for significant integrity guarantees (100 percentage point reduction in hallucination).

Resource Requirements. All experiments ran in Google Colab on an NVIDIA T4 (15GB VRAM), Python 3.10, and MLflow 2.x. Typical end-to-end runtime was ~ 26 minutes for 8 tasks (vs. 24 minutes without governance). API usage costs for LLM calls were minimal relative to compute time and are dominated by the baseline execution cost rather than governance checks.

5.3.5 What This Work Does Not Claim

- **Not claiming:** Prompt-level techniques are useless $\rightarrow$ **Actual claim:** Architectural enforcement necessary *in addition to* prompt engineering.
- **Not claiming:** All research integrity problems solved $\rightarrow$ **Actual claim:** Hallucinated claims in computational workflows eliminated through dual gates.
- **Not claiming:** MLflow is the only viable backend $\rightarrow$ **Actual claim:** Any artifact store with queryable API can serve as the verification substrate.

6 Related Work

6.1 Autonomous Research Systems

AI Scientist [13] generates end-to-end papers: ideas, experiments, and writing. It evaluates output via human reviewers rating paper quality (subjective). It lacks a provenance layer and can't verify claimed results. EviBound adds evidence-bound execution with MLflow verification.

MLR-Copilot [12] is a machine learning research assistant for data processing, model training, and evaluation. It evaluates task completion using F1 scores (continuous metrics) but trusts agent outputs without artifact validation. EviBound introduces a binary hallucination metric with deterministic verification.

6.2 LLM Hallucination Detection

SelfCheckGPT [14] performs consistency-based hallucination detection by sampling multiple outputs and checking consistency. Applied to natural language generation, it requires multiple samples (expensive). EviBound uses single-run verification via external artifacts (deterministic).

FactScore [15] evaluates factuality for long-form text by validating claims against a knowledge base. Applied to biography generation, it relies on static knowledge bases. EviBound validates against execution artifacts (MLflow) instead.

6.3 Governance and Verification

Constitutional AI [2] achieves value alignment through self-critique, using critiques and revisions to align with ethical principles. Its domain is safety and ethics. EviBound applies governance gates to research integrity (orthogonal concern).

Reflexion [20] performs iterative refinement with verbal feedback, where agents reflect on failures and refine actions. However, reflection is verbal rather than artifact-based. EviBound verifies via MLflow API queries (machine-checkable).

6.4 Agent Tool Use and MLOps Verification

ReAct [23] allows reasoning and acting through external tools by combining chain-of-thought with tool calls to improve problem solving. However, it focuses on reasoning quality and does not enforce artifact-level verification.

Toolformer [19] uses self-supervised training for API tool use, teaching LLMs when and how to call APIs. It improves tool competence but not evidence-binding to execution artifacts.

MLOps frameworks (DVC, Weights&Biases) [17, 3] track data, metrics, and artifacts across runs. Governance-oriented data quality and monitoring tools like **Great Expectations** and **Evidently** [8, 6] validate datasets and drift. These are complementary to EviBound: EviBound *binds* claimed results to verifiable artifacts and gates promotion based on API queries.

PAL, MRKL, ToolLLM [7, 10, 18] improve program synthesis and tool routing for LLMs (reasoning + acting). Practical agent frameworks such as **AutoGPT**, **LangChain Agents**, and **CrewAI** [21, 11, 5] orchestrate tools and memory. These works focus on agent competence and orchestration, not on evidence-binding. EviBound is complementary: it governs *promotion* by enforcing approval and verification gates independent of the agent stack.

6.5 Formal Verification Perspective

Formal methods (e.g., model checking) provide strong guarantees in software systems [4]. EviBound shares the spirit of structural guarantees—preventing false promotions by construction—while remaining pragmatic for ML research workflows through API-level evidence checks rather than state-space exploration.

7 Future Work

7.1 Scaling to Complex Multi-Step Research

Future versions should handle more complex, multi-stage research workflows. This means extending the framework to multi-day or distributed experiments such as large-scale training and hyperparameter sweeps. Hierarchical evidence contracts can formally represent sub-task dependencies, ensuring that downstream phases only run once upstream evidence is verified. Incremental verification methods—checkpoint-level validation—would allow recording and evaluating partial progress, improving fault tolerance and traceability for long-running studies.

7.2 Domain-Specific Evidence Schemas

To work across research disciplines, the system needs specialized evidence schemas tailored to domain conventions. In theoretical research, this might mean proof verification via formal systems like Lean or Coq. For systems research, reproducible benchmarking in containerized environments could serve as the evidence standard. In NLP, dataset provenance could be established through authenticated metadata and artifact lineage, like integrations with Hugging Face datasets. These domain-specific schemas would let validation criteria align with accepted practices in each field.

7.3 Adaptive Verification Thresholds

A key next step is adaptive verification thresholds. Instead of a fixed confidence level τ , the system could learn task-specific thresholds from historical success and failure rates. Risk-adaptive gating would let exploratory tasks proceed under more permissive conditions, while production-oriented tasks demand higher confidence. Over time, empirical data could guide this calibration, improving both reliability and throughput without manual tuning.

7.4 Cross-Cycle Learning from Verification Patterns

The system will progressively accumulate knowledge about verification failures and success modes across multiple cycles. The Planning Team can analyze these patterns to identify high-risk task configurations and preemptively apply known corrective measures. A procedural memory cache will be maintained to store stable task templates and common recovery pathways, reducing redundant retries. This cross-cycle feedback loop transforms verification outcomes into actionable planning intelligence, leading to greater efficiency and stability in long-term research execution.

7.5 Planning and Reflection System Integration

Current System Context. Although this paper focuses primarily on the governance and verification processes of the Execution Team, the current implementation already incorporates two complementary sub-systems. The **Planning Team** (comprising four agents) is responsible for generating research tasks with explicit acceptance criteria. Meanwhile, the **Reflection Service** continuously monitors execution and provides corrective patches that inform retry mechanisms and failure analysis.

Future Integration. Planned extensions will deepen the coordination between these components through adaptive task generation and memory-guided governance.

Adaptive Task Generation. The Planning Team will dynamically refine task specifications by learning from historical verification outcomes. Tasks that frequently encounter specific failure modes will have their acceptance criteria adjusted in advance, and high-risk assignments will be pre-patched using previously successful fixes. This adaptation will allow the system to evolve toward increasingly robust task definitions over successive research cycles.

Memory-Guided Governance. The Reflection Service will consolidate experience from prior executions into structured forms of memory—progressing from episodic to semantic and ultimately to procedural knowledge. These procedural memories will act as reusable templates for recurring task types, effectively lowering the retry rate through proactive prevention. Over time, this structured memory will serve as an institutional foundation for consistent, high-quality governance.

Cross-Cycle Learning. A continuous feedback channel will be established between the Reflection Service and the Planning Team. Verification failures and their resolutions will be summarized and transmitted back to inform future planning sessions. Each new cycle will thus incorporate refined constraints and improved task templates derived from accumulated experience. In the long term, this mechanism is expected to yield the emergence of domain-specific governance policies, enabling the system to adapt intelligently to the evolving standards and requirements of different research areas.

Research Opportunity: Characterizing the interplay between planning accuracy, execution governance, and reflection-driven learning as a multi-agent system optimization problem.

8 Conclusion

This paper presents EviBound, an evidence-bound framework that eliminates hallucinated claims through dual governance gates. The results show three things. First, governance changes the integrity equation: hallucinations drop from 100% (no gates) to 25% (verification-only) to 0% (dual gates). Architectural enforcement—not model size—drives trustworthiness. Second, deterministic verification via MLflow API queries binds claims to concrete artifacts and provenance, letting independent readers validate results. Third, these guarantees cost only modest overhead in practice, making governance a pragmatic design choice for real research systems.

The reproducibility package provides:

- Complete execution trajectories for all 8 tasks
- MLflow run_ids for independent artifact validation

- 4-step verification protocol
- Google Colab notebooks (NVIDIA T4 15GB VRAM)

Broader Impact: This work provides a reusable benchmark and an architectural template for trustworthy autonomous research. By showing that integrity needs governance rather than just larger models, this work aims to shift focus toward architectural innovation across the community.

Forward-Looking. I advocate a paradigm shift from prompt-centric tuning to evidence-bound design as the default for autonomous research systems. The community can accelerate this shift by adopting explicit evidence contracts, implementing dual approval/verification gates, and contributing domain-specific schemas and open benchmarks. My vision is a research ecosystem where claims are *born verified*: every result ships with machine-checkable provenance, audits become routine, and agents collaborate safely through shared governance infrastructure. The principle should be universal: *no evidence, no claim*.

9 Reproducibility Package

All experimental artifacts will be made available upon publication:

- **Code Repository:** Complete implementation with execution framework
- **MLflow Tracking Server:** Read-only access for run_id verification
- **Google Colab Notebooks:** All 8 task execution notebooks
- **Claims Ledger:** JSON file with all run_ids and verification timestamps

Availability and License: Code and artifacts will be released under CC BY 4.0 upon publication. Public URLs for the read-only MLflow tracker and artifact bundles will be posted in the repository README upon acceptance; reviewers may request early access via the corresponding author email.

Environment Specifications:

All experiments were conducted in Google Colab with the following environment:

- **Python:** 3.10.12
- **MLflow:** 2.16.2 (tracking server with local artifact storage)
- **Core Dependencies:**
 - anthropic 0.39.0 (Claude API client)
 - torch 2.0.1+cu118 (PyTorch with CUDA support)
 - transformers 4.38.0 (Hugging Face models)
 - sentence-transformers 2.2.2 (embedding models)
 - scikit-learn 1.3.2 (evaluation metrics)
- **Compute:** Google Colab GPU runtime (NVIDIA T4 15GB VRAM)
- **Storage:** MLflow local filesystem backend (no cloud dependency)

Minimal Requirements: Python 3.10+, MLflow 2.16+, 16GB RAM recommended for reproduction. GPU not strictly required but improves execution speed. All dependencies installable via pip with provided requirements.txt.

4-Step Verification Protocol:

1. Clone repository and install dependencies
2. Access MLflow server (read-only credentials provided)
3. Load Claims Ledger (JSON with run_ids)
4. For each verified task:
 - Query: `mlflow.get_run(run_id)`
 - Validate: status, metrics, artifacts
 - Compare with claimed evidence in report

Estimated Runtime and Contact. The full verification protocol typically completes in ~30 minutes (MLflow access: ~5 min, run_id validation: ~20 min, artifact inspection: ~5 min). For questions or access issues, contact rc989@cornell.edu.

Licensing. Code and experimental data will be released under CC BY 4.0; third-party datasets (e.g., Flickr8k) remain subject to their original licenses.

Appendix A: Task Specifications (T01, T03, T04, T05, T06, T09, T12, T13)

For completeness and reproducibility, acceptance criteria and verification requirements are listed requirements for the eight benchmark tasks used in this work. All MLflow checks assume run status is FINISHED and artifacts are logged under the run’s artifact tree.

T01: CLIP Attention Extraction

Artifacts: `attentions/*.npy`, `visualizations/attn_grid.png`.

Verification: Presence of `attentions/` (EviBound: typically 120 files; Baseline B: minimal threshold ≥ 5) and `attn_grid.png`.

T03: Import Error Recovery

Artifacts: `reports/t03_import_recovery.json`.

Verification: JSON file present under `reports/` and readable.

T04: Data Path Validation

Artifacts: `synthetic_fallback_data/` directory (minimum expected files).

Verification: Directory exists under MLflow artifacts; contains expected files.

T05: Results Report Persistence

Artifacts: `reports/results.json`, `reports/summary.md`.

Verification: Both files present under `reports/`; JSON parseable.

T06: Approval Contract Enforcement

Artifacts: `outputs/approval_contract_output.json` with required fields.

Verification: JSON contains keys `{result, confidence, timestamp}`; absence constitutes failure.

T09: API Evidence Check

Artifacts: Valid `run_id` bound to task; required artifacts as specified.

Verification: Missing or invalid `run_id` fails verification even if execution claimed success.

T12: Environment Pinning

Artifacts: `environment/env_metadata.json`.

Verification: File present and includes Python version and dependency snapshot.

T13: Minimal Visualization Export

Artifacts: `visualizations/summary.png` (`analysis_report.md` optional).

Verification: `summary.png` present; optional report may be included for narrative.

Expected Outcome: All 7 VERIFIED_SUCCESS tasks should have queryable `run_ids` with matching metrics and artifacts.

Appendix B: Detailed Phase Mechanics

This appendix provides detailed descriptions of all execution phases referenced in Section 2.

Phase 3: Implementation

Agents propose full code implementations based on the task specification. Each agent contributes an independent approach, focusing on clarity and reproducibility. The goal is disciplined execution, not speculative experimentation.

Phase 4: Approval Gate (Detailed)

The approval gate serves as the first formal validation checkpoint. Before any code runs, the submission must follow a strict acceptance schema with the required fields `run_id`, `metrics`, and `artifacts`. A compliance check ensures these components are present and properly structured. Approval is only granted when all three agents reach unanimous agreement, reinforcing accountability. Successful proposals advance to Phase 5; failed proposals go to Phase 4.5 for revision or get rejected.

Phase 4.5: Approval Retry

When a submission barely misses approval, the bounded retry mechanism allows corrections. Agents can apply patches, but only when their confidence exceeds the threshold τ (typically 0.7). The system allows one or two retries at most. This design encourages careful, evidence-based refinement while preventing infinite retry loops.

Phase 5: Execution

Once approved, the implementation runs in a sandboxed environment that ensures safety and isolation. This controlled setup prevents unintended interference and captures all intermediate results for review. The goal is to get verifiable outputs that match the approved design’s intent.

Phase 5.5: Execution Retry

If runtime failures or incomplete outputs occur, the system starts a limited retry process. Agents diagnose the issue using logs and error traces, then apply targeted fixes before re-running. Each retry gets explicitly documented to maintain transparency and reproducibility.

Phase 6: Verification Gate (Detailed)

After execution completes, the verification gate checks whether outcomes meet integrity standards. Using the MLflow interface, the system confirms the `run_id` exists and is queryable, checks all required artifacts are available, and verifies the execution status equals `FINISHED`. When acceptance criteria define specific metric thresholds, the gate validates those too. Only results that satisfy all conditions move to Phase 7.

Failure Routing: When verification fails, the system routes to the minimal necessary phase for repair based on failure type:

- **Missing `run_id` or artifacts** → Phase 6.5 (evidence regeneration): The execution succeeded but failed to log evidence properly.
- **Execution status not `FINISHED`** → Phase 5.5 (runtime repair): The execution encountered runtime errors or did not complete.
- **Metric violations** → Phase 4.5 (contract refinement): The acceptance criteria were too strict or incorrectly specified.
- **Task scope issues** → Phase 3 (re-planning): The task definition itself needs revision.

This targeted routing prevents wasteful full-cycle retries by directing failures to the earliest phase that can fix the specific problem.

Phase 6.5: Verification Retry

Phase 6.5 specifically handles **evidence regeneration** when the execution succeeded but failed to log artifacts or `run_id` properly. This is distinct from runtime failures (handled by Phase 5.5) or contract issues (handled by Phase 4.5). Agents make a few recovery attempts to regenerate missing evidence—re-logging artifacts, recreating the `run_id`, or fixing MLflow tracking calls. The process stays bounded—usually one or two tries—to ensure remaining failures reflect real methodological issues, not transient logging errors.

Phase 7: Finalize & Reporting

The final phase packages verified results into a structured handoff. This package includes provenance data, links artifacts to their sources, and records the final status. While handoff preparation is more procedural than operational, it’s essential for maintaining transparency, continuity, and reproducibility across research cycles.

References

- [1] Anthropic. Claude 3.5 model family. Anthropic developer documentation. <https://docs.anthropic.com/claude/docs/claude-3-5-models>, 2024. Accessed 2025-10-27.
- [2] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jamie Kerr, Jared Mueller, Jeffrey Wu, Joshua Landau, Kamal Ndousse, Kamile Lukosuite, Liane Lovitt, Michael Sellitto, Nelson Elhage, Nicholas Schiefer, Noemi Mercado, Nova DasSarma, Robert Lasenby,

- Robin Larson, Sam Ringer, Scott Johnston, Shauna Kravec, Sheer El Showk, Stanislav Fort, Tamara Lanham, Timothy Telleen-Lawton, Tom Conerly, Tom Henighan, Tristan Hume, Samuel R. Bowman, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, and Jared Kaplan. Constitutional AI: Harmlessness from AI feedback. *arXiv preprint arXiv:2212.08073*, 2022.
- [3] Lukas Biewald. Weights & biases: Experiment tracking and model management platform. <https://wandb.ai/site>, 2020. Accessed 2025-10-27.
 - [4] Edmund M Clarke, Orna Grumberg, and Doron A Peled. *Model Checking*. MIT Press, 1999.
 - [5] CrewAI. Crewai: Multi-agent framework. <https://docs.crewai.com/>, 2024. Accessed 2025-10-27.
 - [6] Evidently AI. Evidently AI: Monitoring and testing for machine learning. <https://www.evidentlyai.com/>, 2024. Accessed 2025-10-27.
 - [7] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. *arXiv preprint arXiv:2211.10435*, 2022.
 - [8] Great Expectations. Great expectations: Data quality for analytics and ML. <https://greatexpectations.io/>, 2024. Accessed 2025-10-27.
 - [9] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Towards mitigating hallucination in large language models via self-reflection. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023.
 - [10] Ehud Karpas, Omri Abend, Yonatan Belinkov, Barak Lenz, Opher Lieber, Nir Ratner, Yoav Shoham, Hofit Bata, Yoav Levine, Kevin Leyton-Brown, Dor Muhlgay, Noam Rozen, Erez Schwartz, Gal Shachaf, Shai Shalev-Shwartz, Amnon Shashua, and Moshe Tenenbholz. MRKL systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning. *arXiv preprint arXiv:2205.00445*, 2022.
 - [11] LangChain. Langchain agents documentation. <https://python.langchain.com/docs/modules/agents/>, 2024. Accessed 2025-10-27.
 - [12] Ruochen Li, Teerth Patel, Qingyun Wang, and Xinya Du. MLR-Copilot: Autonomous machine learning research based on large language models agents. *arXiv preprint arXiv:2408.14033*, 2024.
 - [13] Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The AI scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.
 - [14] Potsawee Manakul, Adian Liusie, and Mark J. F. Gales. SelfCheckGPT: Zero-resource black-box hallucination detection for generative large language models. *arXiv preprint arXiv:2303.08896*, 2023.
 - [15] Sewon Min, Kalpesh Krishna, Xinxu Lyu, Mike Lewis, Wen-tau Yih, Pang Wei Koh, Mohit Iyyer, Luke Zettlemoyer, and Hannaneh Hajishirzi. FActScore: Fine-grained atomic evaluation of factual precision in long form text generation. *arXiv preprint arXiv:2305.14251*, 2023.
 - [16] MLflow Contributors. MLflow: An open source platform for the machine learning lifecycle. <https://mlflow.org/docs/latest/index.html>. Accessed 2025-10-27.
 - [17] Dmitry Petrov et al. DVC: Data version control. <https://dvc.org/>, 2020. Accessed 2025-10-27. Open-source data and experiment version control system by Iterative.ai.
 - [18] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. *arXiv preprint arXiv:2307.16789*, 2023.

- [19] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.
- [20] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023.
- [21] Significant Gravitas. Auto-GPT: An autonomous GPT-4 experiment. <https://github.com/Significant-Gravitas/AutoGPT>, 2023. Accessed 2025-10-27.
- [22] S. M. Towhidul Islam Tonmoy, S. M. Mehedi Zaman, Vinija Jain, Anku Rani, Vipula Rawte, Aman Chadha, and Amitava Das. A comprehensive survey of hallucination mitigation techniques in large language models. *arXiv preprint arXiv:2401.01313*, 2024.
- [23] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023. arXiv:2210.03629.
- [24] Wan Zhang and Jing Zhang. Hallucination mitigation for retrieval-augmented large language models: A review. *Mathematics*, 13(5):856, 2025.