European Journal of Computer Science and Information Technology, 14(8),1-12, 2026
Print ISSN: 2054-0957 (Print)
Online ISSN: 2054-0965 (Online)
Website: https://www.eajournals.org/
Publication of the European Centre for Research Training and Development -UK

Tool-Layer Validation Gaps in Autonomous Large Language Model Agent Systems

Basel Atiyire*
Western Illinois University, USA
ORCID: 0009-0003-6869-933X

Ransom Igodi
Western Illinois University, USA
ORCID: 0009-0008-1411-8946

Japheth Kodua Wiredu
Regentropfen University College, Upper East Region, Ghana
ORCID: 0009-0008-0313-5011

doi: https://doi.org/10.37745/ejcsit.2013/vol14n8112 Published November 08, 2026

Citation: Atiyire B., Igodi R. and Wiredu J.K. (2026) Tool-Layer Validation Gaps in Autonomous Large Language Model Agent Systems, *European Journal of Computer Science and Information Technology*, 14(8),1-12

Abstract: *Language model agents that write to live databases carry a risk that read-only systems do not: an invalid input that passes the tool layer can silently corrupt data the organization depends on. This paper empirically examines where validation sits in the tool stack, using two tool-access routes that coexist in one deployed human resources agent: a direct route with minimal checking and a service-mediated route that enforces organizational rules before writing. Across 49 edge cases, the direct route silently corrupted the database in 14 cases that the service layer rejected; both routes rejected 14; both accepted the same bad input in 12, revealing a shared blind spot; and nine reflected mismatched operations between the routes. The findings suggest that data integrity depends less on model reasoning than on where and how complete validation logic is placed. A service layer helps substantially, but alone it is not a complete defense.*

Keywords: Language model agents, tool use, data integrity, service layer design, write operations, agentic systems

INTRODUCTION

Language model agents equipped with tools have moved quickly from answering questions to actually doing things like hiring an employee, approving leave, provisioning a laptop, kicking off onboarding all through plain natural language instructions that commit changes directly to a live database (Dasaklis et al., 2025). That shift matters more than it might first appear. A wrong answer to a question is just a wrong answer; a wrong write to a database becomes a fact the rest of the organization now has to live with. A duplicate employee record, a leave request approved by someone who shouldn't have been able to approve it, an invalid role quietly assigned these are the kinds of errors that don't announce themselves. They surface later, in a payroll discrepancy or an access-control incident, often long after anyone could trace the error back to its source.

One of the more consequential decisions a team makes when building a system like this is where, exactly, the organization's rules get enforced. Some tool implementations invoke the database more or less directly, with little standing between the agent's request and the write itself. Others route every call through an intermediate layer, a set of service classes that checks referential integrity, role permissions, and valid state transitions before anything touches storage. As far as we can tell, no prior study has actually measured, empirically and in a live system, what these two approaches produce when tested side by side. We used a human resources platform as the setting for this study, but the underlying issue isn't really about human resources at all; it's a property of how tools get called, and it applies to any agent writing to any structured database.

This paper walks through a study of one such platform, where both approaches happen to coexist in the same codebase. We built 49 edge-case inputs, ran everyone to a clear and repeatable conclusion, and organized the rest of the paper as follows. The Literature Review and Theoretical Underpinning section situates the study against prior research on agent tool use, agent security, service-layer design, and transactional integrity. The Methodology section describes the system and how the tests were run. Results and Findings report what we found, broken down by category. The Discussion works through what the findings mean for how these systems should be built, and Implications for Research and Practice turn them into a starting checklist for tool design. The Conclusion states the paper's core contribution, and Future Research outlines where this line of work might go next.

LITERATURE REVIEW AND THEORETICAL UNDERPINNING

Agents that use tools

Surveys of language model agents treat tool use as part of the agent's action module, the component that lets it act on its environment rather than only describe it (Wang et al., 2024), and the tool-learning literature is commonly organized around four stages: task planning, tool selection, tool calling, and response generation (Qu et al., 2025). Early work on agent reasoning, most notably the ReAct framework, showed that interleaving reasoning with action lets an agent complete multi-step tasks through repeated tool calls (Yao et al., 2023). Later systems such as Toolformer and Gorilla showed that giving an agent a well-structured tool definition cuts down on made-up or malformed parameters (Schick et al., 2023; Patil et al., 2023), and larger efforts such as ToolLLM, built on more than 16,000 real-world APIs (Qin et al., 2024), and API-Bank, which evaluates how well models plan, retrieve, and call 73 API tools (Li et al., 2023), have pushed tool-use capability further.

Benchmarks such as AgentBench have since given the field a standard way to measure how well an agent completes multi-step tasks across different environments (Liu et al., 2024). The closest to our setting is τ-bench, which gives an agent domain-specific API tools and a written domain policy, and judges success by comparing the database state at the end of a conversation with an annotated goal state (Yao et al., 2024). Even there, though, the policy is placed in the agent's system prompt, so following the rules remains the model's responsibility rather than something the tools enforce. Taken together, this work is about whether the agent picks the right tool, follows the rules, and completes the task, not about what the tool itself actually checks once it's been called.

Keeping agents' safe

A separate body of work looks at prompt injection (Greshake et al., 2023), agents being given more autonomy than is safe (Anderljung et al., 2023), and adversarial manipulation of what a tool returns

(Perez and Ribeiro, 2022). Hou et al. (2025), for instance, survey the security landscape around the Model Context Protocol. Benchmarks have since made these risks measurable for tool-using agents. InjecAgent tests indirect prompt injection across 1,054 cases and found a ReAct-prompted GPT-4 agent vulnerable 24% of the time (Zhan et al., 2024), and AgentDojo provides an extensible environment for evaluating such attacks and defenses on realistic agent tasks (Debenedetti et al., 2024). ToolEmu uses a language model to emulate tool execution and flag risky agent actions, reporting that even the safest agent it tested failed 23.9% of the time (Ruan et al., 2024), while R-Judge measures whether models can recognize safety risks in records of agent interactions (Yuan et al., 2024).

Practitioner guidance points in a similar direction. The OWASP Top 10 for LLM Applications lists excessive agency as a distinct vulnerability and recommends implementing authorization in downstream systems rather than relying on the model to decide whether an action is allowed (OWASP, 2025). Most of the research above, however, addresses risk at the level of the agent itself: whether it can be manipulated, and whether it recognizes a dangerous action. The OWASP guidance is the exception, but it is a recommendation rather than a measurement. What we look at here is different: silent corruption that happens not because the agent misbehaves, but because the tool it calls was never built to say no.

A familiar pattern from software engineering

Putting business rules in a dedicated layer, separate from the raw database access, is not a new idea; it is a long-standing practice in software architecture, familiar from the repository pattern and domain-driven design. Recent work on transduction for structured data workflows makes a related point from a different angle, arguing that structured data pipelines need their own dedicated handling rather than being treated as an afterthought to model reasoning (Gliozzo et al., 2025). Security guidance for language model applications adds the principle of complete mediation, under which every request an agent makes to a downstream system is checked against policy rather than trusted because the model issued it (OWASP, 2025).What we do here is take these principles and test them empirically against language model agent pipelines specifically, where the boundary between application logic and the database is now mediated by a tool call rather than an ordinary method call. The caller of a tool is now a language model that cannot be relied on to guarantee that its inputs are valid, which places the burden of checking on the tool itself, and a second route to the database that skips the service layer is a path around complete mediation.

Consistency across multi-step plans

Other recent work, such as SagaLLM (Chang and Geng, 2025), looks at whether a multi-step plan holds together and stays consistent across several agents, combining the saga transactional pattern with automated compensation and independent validation agents. That's a related but distinct question from the one we're asking. We're not asking whether a plan is feasible; we're asking whether each individual call, on its own, enforces the rule it's supposed to.

METHODOLOGY

The system under study

This study uses a human resources agent that was already running in production, deployed on cloud infrastructure and backed by a lightweight file-based database. Twelve tools cover the core workflows:

hiring, leave management, technical support provisioning, onboarding, and ticket handling.Two separate ways of reaching the database exist side by side in the same codebase. The first calls the database more or less directly, through raw tool definitions built into the agent's reasoning loop, with very little checking along the way. The second routes the identical operations through a set of service classes one each for employees, leave, tickets, and onboarding that enforce the organization's actual rules before any record is written. Both ultimately write to the same underlying database. Figure 1 shows how the two routes relate to each other.

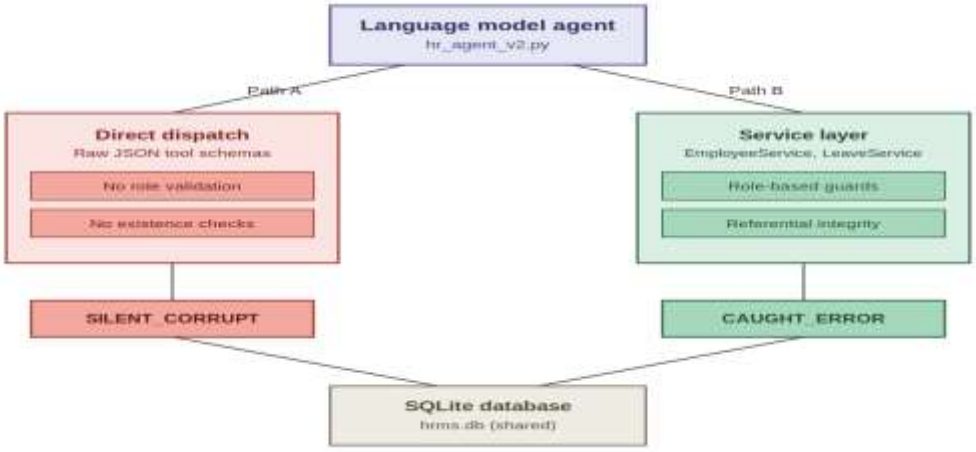


Figure 1.Two ways of reaching the same database: a direct route with minimal checking, and a service-mediated route that enforces the organization's rules before anything is written.

What each route actually checks

Table 1 lists the checks that the service-mediated route performs but the direct route does not. These gaps weren't designed in deliberately; they emerged naturally as the system was built with the help of AI-assisted software development, which arguably makes them more representative of what tends to happen in practice rather than less.

Table 1. Which rules each route actually enforces?

Rule Being Checked	Direct Route	Service-Mediated Route
System role must come from an approved list	Not checked	Checked
Email address must be unique	Not checked	Checked
A manager cannot be their own manager	Not checked	Checked
Referenced manager must actually exist	Not checked	Checked
Email must be properly formatted	Not checked	Checked
Leave approver must actually exist	Not checked	Checked
Overlapping leave dates, at creation	Tool not available on this route	Checked
Ticket's employee must actually exist	Not checked	Checked
Onboarding task must actually exist	Not checked	Checked
Task status must be a valid value	Not checked	Checked

Designing and running the tests

We built 49 edge cases across five categories: hiring (10), ticket creation (five), leave management (18), onboarding (six), and general state consistency (10). Each one was chosen to probe a specific

rule that one or both routes might or might not enforce. Figure 2 walks through one representative example.

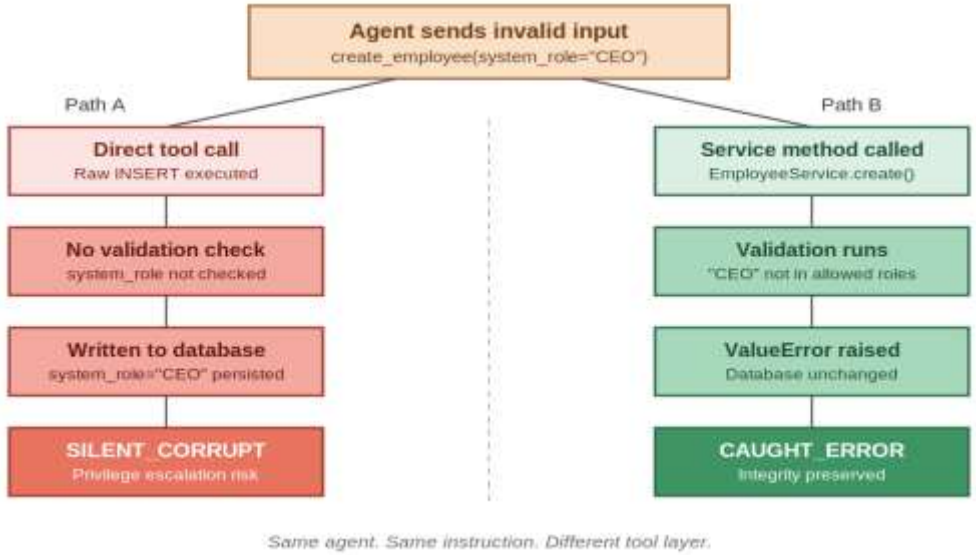


Figure 2. The same instruction sent down both routes: the direct route writes the bad value silently, while the service-mediated route catches it before the database is touched.

Every input was run against both routes using a freshly reset copy of the starting database. We initially ran the tests in a single ongoing process, which produced spurious errors on 12 of the 49 cases not because of anything wrong with the system itself, but because database connections were quietly accumulating across sequential calls within that one process. Once we caught this and switched to running every test in its own fully isolated subprocess, with the database reset before each individual call, all 49 tests produced a clean, repeatable result with nothing left unresolved. Each outcome was classified one of three ways: the input was silently accepted and an invalid record written, the input was correctly rejected with the database left unchanged, or the relevant tool simply wasn't available on that route. No calls to a language model were made during testing; only the tool-execution layer itself was exercised, which let us isolate the effect of the architecture from anything to do with how well or poorly the model reasons.

Limitations

Three things bound how far these findings generalize. First, this is a single system in a single domain, so broader claims would need replication across other domains and other frameworks. Second, the two-route pattern we studied wasn't something we engineered on purpose; it emerged organically as the system evolved through AI-assisted software development. That means the specific gaps catalogued here reflect how a real system actually turned out, not a controlled experiment where we deliberately varied where the checks sit. Third, and deliberately, this paper stops at diagnosis. We measure and characterize where validation gaps occur and how large they are, but we do not propose, build, or test a fix, a validation framework, or a mitigation pattern. We made this choice because we think an accurate, empirically grounded picture of the problem needs to exist before a proposed solution can be evaluated against it. A mitigation strategy tested only against gaps the same team invented would be a weaker contribution than one tested against gaps documented independently first.

Building and evaluating such a framework is substantial work in its own right, which is why we treat it as a direction for future research rather than folding it into this paper.

RESULTS AND FINDINGS

The overall picture

Table 2 breaks down what happened across all 49 tests, every one of them resolved cleanly under the isolated protocol. Of the nine cases reflecting a mismatch in which tools were even available, eight were cases where the direct route simply had no tool for creating a leave request at all, and one ran the other way the service-mediated route had no way to close a ticket, something the direct route could do.

Table 2. How all 49 test cases played out.

Outcome	Count	Share of 49 Tests
Direct route corrupts the data; service-mediated route catches it	14	28.6%
Both routes correctly reject the input	14	28.6%
Neither route catches it — a shared blind spot	12	24.5%
Direct route is missing a tool the other route has	8	16.3%
Service-mediated route is missing a tool the other route has	1	2.0%
Total	**49**	**100%**

Where the direct route fails but the service layer catches it

Table 3 breaks down the 14 cases where the service-mediated route caught something the direct route missed entirely. Hiring accounts for five of these, the largest single category, which makes sense given how many rules apply at the moment someone is hired. Leave management contributes two: accepting a leave approver or rejecter who doesn't actually exist, both of which leave a false trail in the audit record.

Table 3.Cases where the direct route silently let something through and the service-mediated route correctly stopped it.

Case	Category	What Wasn't Checked	What Could Go Wrong
A01	Hiring	An invalid system role was accepted	A person could gain more access than intended
A03	Hiring	A duplicate email address was accepted	Two records collide on the same identity
A04	Hiring	A manager was allowed to be their own manager	A circular reporting line
A05	Hiring	A manager who doesn't exist was accepted	A reference pointing at nothing
A09	Hiring	A malformed email address was accepted	Notifications silently fail to deliver
B01	Ticket	A ticket was created for an employee who doesn't exist	An orphaned ticket with no owner
B05	Ticket	A malformed employee ID was accepted	A ticket that can never be resolved
C09	Leave	A nonexistent approver was accepted	A false entry in the audit trail
C12	Leave	A nonexistent rejecter was accepted	A false entry in the audit trail

Case	Category	What Wasn't Checked	What Could Go Wrong
D01	Onboarding	Onboarding was started for an employee who doesn't exist	Orphaned task records
D03	Onboarding	A write to a task that doesn't exist appeared to succeed	A false record of completion
D04	Onboarding	A write for an employee who doesn't exist appeared to succeed	A false record of completion
D06	Onboarding	An invalid task status string was accepted	The task is left in an undefined state
E04	Consistency	Overlapping leave wasn't re-checked at approval time	An employee ends up double-booked

The shared blind spot

Twelve cases revealed something more concerning than a gap between the two routes: inputs that neither route caught. That's a much larger category than you'd expect if service-layer mediation alone were sufficient protection. These included empty or missing required fields (an empty system role, an empty name, an empty ticket category, an empty leave reason); unusual characters in a text field that were stored safely, thanks to parameterized queries, but never actually flagged as suspicious input; an unauthorized privilege being granted, where both routes accepted a request to assign the human resources administrator role with no check on whether the requester was allowed to grant it; duplicate tickets for the same employee and the same item going unnoticed by either route; an unreasonably long name (500 characters) accepted without any length limit; onboarding tasks generated for an employee whose status was already marked inactive; and, as one exception worth noting, both routes correctly accepting names with unicode and mixed-script characters, which is actually the right behavior and included here only for completeness rather than as a defect.

What we found on role-based permissions

Leave approval and rejection looked, at first glance, like a clean pass for both routes every role-inappropriate attempt was correctly blocked once we tested the full lifecycle of a request. But the reason turned out to matter a great deal. The service-mediated route blocked these attempts because it actually checks who's allowed to approve or reject. The direct route blocked them too, but only because, by the time our test tried the bypass, the leave request had already moved out of a pending state, an accident of test sequencing, not a real safeguard. It is important to be precise about what this means: we did not execute a test that isolates a pending-state request and attempts an unauthorized approval through the direct route, so no test result directly demonstrates the bypass. What we did do is inspect the direct route's tool definition directly, and confirm that it contains no role parameter at all for approving or rejecting leave. Based on that inspection, a leave request still sitting in a pending state, approved through the direct route by someone with no approval authority, would be expected to go through without any check. We present this as a risk identified through code inspection rather than through an executed test, and flag it as a specific case that a follow-up test should directly target rather than infer.

When the two routes simply don't match

Eight of our tests confirmed that the direct route has no way to create a leave request at all consistent with what we had already observed informally while building and testing this system, where leave management through that route never completed successfully. One further case showed the mismatch running in the opposite direction: the direct route can close a ticket, but the service-mediated route has

no equivalent capability, meaning that interface simply cannot close tickets at all. A mismatch like this is itself a kind of validation gap. The safest thing to do when an operation isn't implemented is to expose no tool for it at all but a system offering two different ways to reach the same data probably shouldn't let those two paths quietly diverge in what they're each capable of doing.

DISCUSSION

None of the gaps documented here come down to the agent making a bad call. In every single case, the agent did exactly what it was asked to do. The problem sat downstream, in the tool that received the instruction and simply wasn't built to say no. The blind-spot findings push this even further: even when the right checking logic exists somewhere in the system, its coverage can still have real holes, and a service layer only protects against what its authors thought to write checks for.

Most agent frameworks in use today treat a tool as a fairly thin wrapper: a name, a schema, and a call out to some external system or database. What this study suggests is that the boundary needs to be thicker than that. It should carry the actual rules that keep the underlying data consistent, and just as importantly, it needs to be checked systematically against a real list of edge cases, rather than simply trusted to be complete because a service layer happens to exist. This is consistent with the OWASP recommendation to enforce authorization downstream rather than leaving it to the model (OWASP, 2025); our results add a measurement of what happens when that recommendation is only partly followed.

We want to be direct about what this paper does and doesn't claim to do. It diagnoses where and how validation gaps occur, and measures how large they are; it does not build or empirically test a way to close them. We see that as the right order of operations rather than a shortcoming to apologize for a mitigation approach is only as credible as the problem set it's tested against, and we wanted that problem set to be grounded in real, independently observed gaps before anyone, including us, starts building a fix around it. What we can offer, short of a tested framework, is a concrete starting point drawn directly from what we found, set out in the next section.

IMPLICATIONS FOR RESEARCH AND PRACTICE

Implications for research

For research, the findings suggest that evaluations of tool-using agents should look beyond whether the agent selects the right tool and completes the task, which is where benchmarks such as AgentBench and τ-bench concentrate (Liu et al., 2024; Yao et al., 2024), and should also test what the tool layer itself accepts or rejects. Because no language model calls were made during testing, every gap reported here is attributable to the tool layer rather than to model behavior, so better model reasoning on its own cannot be counted on to close them.

Implications for practice: toward a tool-layer validation contract

Looking across all 26 cases where at least one route let something through it shouldn't have (the 14 enforcement gaps and the 12 shared blind spots together), a pattern emerges: almost every one of them falls into one of five checkable categories. We propose these as a starting checklist: a validation

contract that a tool definition could reasonably be expected to satisfy before it's trusted to write to a database. This is a design proposal drawn from our findings, not a framework we built or tested; we present it as a concrete next step rather than a verified solution.

Identity and reference checks: does every foreign-key-style reference in the input (a manager, an approver, an employee, a task) actually point to a record that exists? Five of our gap cases and two blind-spot cases trace directly to a missing check of this kind.

Authorization-to-grant checks: when an input requests a privileged role or state change, is there a check on whether the requester is allowed to grant it, separate from whether the value itself is well-formed? Our A01 and A10/E05 cases both slipped through because this distinction wasn't enforced.

Format and completeness checks: are required fields actually present, non-empty, and in the expected shape (a valid email, a bounded string length, a valid enum value)? This single category accounts for roughly a third of the blind-spot cases we found.
State-consistency checks, re-verified at every transition: a rule checked once at creation, like leave overlap, needs to be checked again at every later state change, not assumed to still hold. Our E04 case shows what happens when this re-check is skipped.

Symmetric tool coverage across routes: if two interfaces exist for the same underlying data, do they expose the same operations? Our asymmetric-availability cases show that a missing tool is itself a silent validation gap, not just an inconvenience.

CONCLUSION

We studied a human resources agent where two different ways of reaching the database happen to coexist, and compared how each one handled a systematically designed set of edge cases. The domain here happens to be human resources, but the pattern isn't specific to it. The exact field names would change from one system to the next, but the underlying failure of a tool that accepts something it really should have rejected shows up anywhere an agent is writing to a structured database.The core contribution of this paper is empirical evidence that where validation sits in the tool stack shapes data integrity outcomes on its own, independent of how well the underlying model reasons, and that even a well-intentioned service layer only offers partial protection, not complete protection.

Future Research

We see real value in future work that takes the five-category checklist proposed above, implements it as an actual enforcement layer, and empirically re-tests it against a fresh set of edge cases ideally on a different system than the one studied here, to check whether the pattern generalizes. That is the natural next study, and it is deliberately not this one.

Future work could also replicate this study across other domains and other agent frameworks, build tooling that automatically checks service-layer coverage against a systematic list of edge cases, and explore whether instructions given to the model itself might offer an additional line of defense for the

blind-spot category that architecture alone doesn't cover. A follow-up test should also directly target the leave-approval bypass that this study identified through code inspection rather than through an executed test.

Author Contributions
All authors have reviewed the final version to be published and agreed to be accountable for all aspects of the work.

Concept and design: Basel Atiyire
Acquisition, analysis, or interpretation of data: Basel Atiyire

Drafting of the manuscript: Basel Atiyire

Critical review of the manuscript for important intellectual content: Ransom Igodi, Japheth Kodua Wiredu

Methodology and test design review: Ransom Igodi
Supervision: Basel Atiyire

DECLARATIONS
Conflicts of interest: The authors declare that they have no financial relationships, at present or within the previous three years, with any organization that might have an interest in the submitted work, and no other relationships or activities that could appear to have influenced the submitted work.

Funding: This work received no external or specific grant funding, and no financial support was received from any organization for the submitted work.

Ethical considerations: This study evaluated a software system's automated tool-execution layer using synthetic test inputs against a seed database. No human participants, patient data, or personally identifiable information were involved, and no ethics board approval was required.

Data availability: The test dataset consists of 49 systematically designed edge-case inputs spanning five workflow categories (employee creation, ticket creation, leave management, onboarding, and state consistency), along with the per-test outcome classification (silently accepted, correctly rejected, or tool unavailable) for both tool-access routes evaluated in this study. The data and code supporting this study are openly available in the GitHub repository at github.com/BaselAtiyire/hrms-agent.

Code availability: The repository contains the deployed human resources agent (hr_agent_v2.py), the direct-dispatch and service-layer tool implementations evaluated in this study, and the isolated execution harness used to run all 49 test cases as independent subprocesses against a freshly reset seed database. The core function of the harness is to submit each edge-case input to both tool-access routes and classify the resulting outcome. The approach is a black-box, tool-execution-level test: no language model API calls are made during testing, isolating architectural effects from model behavior. Key dependencies include Python and SQLite; installation and usage instructions are provided in the

repository's documentation. The repository is publicly available on GitHub for community review and reuse, subject to the license specified in the repository.

REFERENCES

Anderljung, M. et al. (2023) *Frontier AI regulation: Managing emerging risks to public safety*, arXiv preprint arXiv:2307.03718.

Chang, E. Y. and Geng, L. (2025) *SagaLLM: Context management, validation, and transaction guarantees for multi-agent LLM planning*, Proceedings of the VLDB Endowment, 18(12), 4874-4886.

Dasaklis, T. K. et al. (2025) *Large language models in human resource management: A systematic literature review*, SSRN Preprint, doi:10.2139/ssrn.5314976.

Debenedetti, E. et al. (2024) *AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents*, In Advances in Neural Information Processing Systems 37 (NeurIPS 2024), Datasets and Benchmarks Track, arXiv:2406.13352.

Gliozzo, A. M. et al. (2025) *Transduction is all you need for structured data workflows*, arXiv preprint arXiv:2508.15610.

Greshake, K. et al. (2023) *Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection*, arXiv preprint arXiv:2302.12173.

Hou, X. et al. (2025) *Model Context Protocol (MCP): Landscape, security threats, and future research directions*, arXiv preprint arXiv:2503.23278.

Li, M. et al. (2023) *API-Bank: A comprehensive benchmark for tool-augmented LLMs*, In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP), Singapore, pp. 3102-3116.

Liu, X. et al. (2024) *AgentBench: Evaluating LLMs as agents*, In Proceedings of the International Conference on Learning Representations (ICLR), Vienna, Austria, arXiv:2308.03688.

OWASP (2025) *LLM06:2025 Excessive Agency*, OWASP Top 10 for LLM Applications 2025, OWASP Gen AI Security Project. Available at: https://genai.owasp.org/llmrisk/llm062025-excessive-agency/ (Accessed: 25 October 2026).

Patil, S. G. et al. (2023) *Gorilla: Large language model connected with massive APIs*, arXiv preprint arXiv:2305.15334.

Perez, F. and Ribeiro, I. (2022) *Ignore previous prompt: Attack techniques for language models*, arXiv preprint arXiv:2211.09527.

Qin, Y. et al. (2024) *ToolLLM: Facilitating large language models to master 16000+ real-world APIs*, In Proceedings of the International Conference on Learning Representations (ICLR), Vienna, Austria, arXiv:2307.16789.

Qu, C. et al. (2025) *Tool learning with large language models: A survey*, Frontiers of Computer Science, 19(8), 198343, doi:10.1007/s11704-024-40678-2.

Ruan, Y. et al. (2024) *Identifying the risks of LM agents with an LM-emulated sandbox*, In Proceedings of the International Conference on Learning Representations (ICLR), Vienna, Austria, arXiv:2309.15817.

Schick, T. et al. (2023) *Toolformer: Language models can teach themselves to use tools*, In Advances in Neural Information Processing Systems (NeurIPS), New Orleans, USA, arXiv:2302.04761.

Wang, L. et al. (2024) *A survey on large language model based autonomous agents*, Frontiers of Computer Science, 18(6), 186345, doi:10.1007/s11704-024-40231-1.

Yao, S. et al. (2023) *ReAct: Synergizing reasoning and acting in language models*, In Proceedings of the International Conference on Learning Representations (ICLR), Kigali, Rwanda, doi:10.48550/arXiv.2210.03629.

Yao, S. et al. (2024) *τ-bench: A benchmark for tool-agent-user interaction in real-world domains*, arXiv preprint arXiv:2406.12045.

Yuan, T. et al. (2024) *R-Judge: Benchmarking safety risk awareness for LLM agents*, In Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, USA, pp. 1467-1490.

Zhan, Q. et al. (2024) *InjecAgent: Benchmarking indirect prompt injections in tool-integrated large language model agents*, In Findings of the Association for Computational Linguistics: ACL 2024, Bangkok, Thailand, pp. 10471-10506.