AI SYSTEMS

Tool-Using LLM Agents in Production Business Workflows: A Practitioner Review

A practitioner review of tool-using LLM agents in production: constrained tool design, human-in-the-loop gates, failure modes and evaluation, grounded in a live agency platform.

Bhaskar Roy Sarkar, cazyweb · July 17, 2026 ·

LLM agents | tool use | function calling | human-in-the-loop | prompt injection | AI reliability

Download PDF

ABSTRACT

Large language models became economically interesting to operators the moment they could act: query a database, update a record, write a file, and chain those steps toward a goal. Yet the literature on tool-using agents is dominated by benchmark results, while the decisions that determine whether an agent survives contact with a real business - which tools it gets, where validation lives, where a human sits - are documented mostly in scattered engineering folklore. This paper reviews the research foundations of tool use in language models, from ReAct-style reasoning-acting loops and self-supervised tool learning through modern function-calling interfaces, and connects them to the architectural patterns that production deployments converge on: constrained tool surfaces, authorization enforced below the model, deterministic code for reliability-critical work, and draft-only outputs at expensive boundaries. We ground the review in first-party observations from an agency operations platform that runs six production agents,

and we synthesize the security and evaluation literature - hallucinated actions, indirect prompt injection, cost-blind benchmarks - into concrete guidance. The recurring theme is that agent reliability is an architecture property before it is a model property.

1. Introduction

A language model that only generates text is a drafting tool. A language model that can call functions - look up a record, mutate state, write a file, invoke another program - is a worker, and workers need management. The gap between those two framings is where most production failures live.

The research literature has moved fast on the capability side. We now have a well-understood loop in which a model interleaves reasoning with actions and observations[1], models that teach themselves when to call APIs[2], and systems trained to navigate thousands of real-world tools[3]. What has lagged is the operational side: published guidance on how tool-using agents should be constrained, supervised, and evaluated when they act on real business state, for real customers, with real money attached. Surveys of the field note the same imbalance - rich taxonomies of agent architectures, thin coverage of deployment discipline.[4][5]

This paper is a practitioner review. It has three goals. First, to summarize the research lineage of tool use in language models accurately enough that a practitioner knows which properties are established and which are marketing. Second, to describe the architectural patterns that production systems converge on, using the agent platform we operate - six agents embedded in a live agency operations system - as a running case study. Third, to connect the security and evaluation literature to concrete design rules. Our claims about our own systems are architectural descriptions, not performance claims; we report how the systems are built, because that is the part that generalizes.

2. From Text Generators to Tool Users

Three research threads converged to produce the modern tool-using agent.

The first is the reasoning-acting loop. ReAct demonstrated that interleaving chain-of-thought reasoning with actions against an environment, and feeding the observations back into the context, outperforms either reasoning or acting alone, and reduces hallucination by letting the model check the world instead of imagining it.[1] This loop - plan, act, observe, repeat - remains the skeleton of essentially every production agent, whatever the vendor calls it.

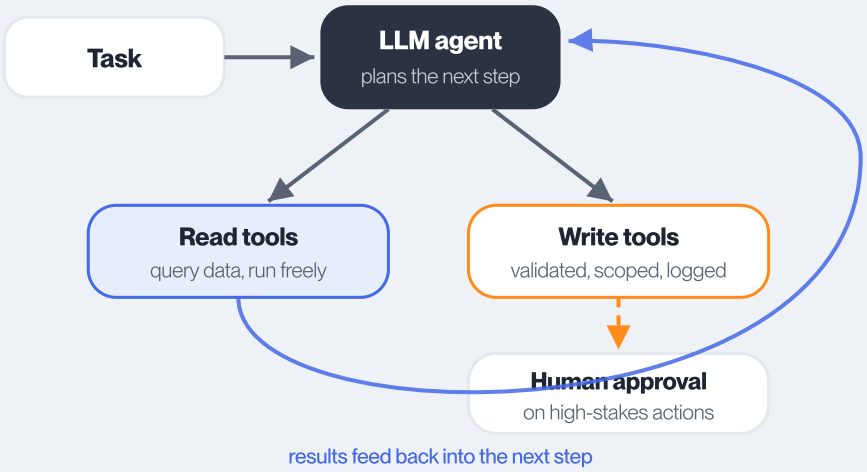
The second thread is tool learning itself. Toolformer showed a model can learn, self-supervised, when to call external APIs (a calculator, a search engine, a translator) and how to fold the results into generation.[2] Gorilla and ToolLLM scaled the idea to large real API surfaces, training and evaluating models on accurate invocation of thousands of endpoints.[6][3] The commercial descendants of this work are the structured function-calling interfaces every major provider now ships: the developer declares a typed schema per tool, and the model emits a validated call rather than free text. The survey literature frames all of this as 'augmentation' - delegating what the model is bad at (arithmetic, fresh facts, side effects) to systems that are good at it.[5]

The third thread is self-improvement within an episode. Reflexion showed that an agent that critiques its own failed attempt in natural language and retries with that critique in context improves markedly without any weight updates.[7] In production this appears as revision loops: an agent that produces an artifact, receives structured feedback (from a validator, a reviewer, or a user), and iterates.

By 2024 the practitioner synthesis had caught up with the research. Anthropic's engineering guidance distinguishes workflows (LLM steps orchestrated by deterministic code) from agents (the model directs its own tool loop), and argues for the simplest composition that works - a position our operating experience strongly supports.[8]

3. Anatomy of a Production Tool Loop

A production agent is the ReAct loop plus an envelope of controls. The model receives a task and a set of tool schemas; it emits either a tool call or a final answer; the runtime executes the call, appends the result to the context, and re-invokes the model. The envelope is where production diverges from the papers:



The production tool-use loop: the model plans, calls a constrained tool, observes the result, and repeats. Read tools execute freely; state-changing tools pass through validation and, for high-stakes actions, a human approval gate.

Iteration caps. Every loop needs a hard stop. Our operations assistant is capped at eight tool-calling turns per request; on hitting the cap it reports what it accomplished rather than continuing silently. Caps convert runaway behavior from an incident into a bounded, observable outcome.

Typed schemas with server-side validation. The model's arguments are parsed against the declared schema, then re-validated by the tool implementation itself: identifiers must resolve, dates must parse, enumerated values must belong to the actual set for that project. The prompt is never the last line of defense.

Observation budgets. Tool results are truncated and paginated (our task-search tool returns at most twenty rows) because context is a finite resource

and long contexts degrade retrieval: models attend best to the beginning and end of the window, with measurable loss in the middle.[9] An agent drowning in its own observations gets worse, not smarter.

Audit emission. Every state-changing call emits the same activity-log entry and real-time event a human edit would. The agent is a principal in the system, not a backdoor around it.

4. Constrained Tool Surfaces and Least Privilege

The single most consequential design decision is what tools the agent does not get. Our platform runs six agents, and their tool surfaces illustrate the spectrum:

The **operations assistant** holds twenty tools - six reads, nine writes, five analytics queries - and nothing else: no filesystem, no shell, no network. Its write tools re-check the requesting user's authorization at the data layer before every mutation, so a model decision can never exceed the human's own permissions. Authorization enforced below the model is categorically stronger than authorization requested in the prompt, because it holds regardless of what the model generates.

The **design and code agents** are file workers. Each runs inside a staged session directory pre-populated by deterministic code with everything the task needs - the brief, brand context, a faithful static snapshot of the client's live page, screenshots, stylesheets - and holds exactly four capabilities: read, glob-search, grep-search, and write, scoped to that directory. They cannot fetch a URL, execute a command, or see the rest of the machine. Anything that requires reliability or touches the outside world (scraping the page, resolving assets, taking screenshots) happens in ordinary code before the agent starts. This inversion - shrink the agent's world, move risk into deterministic pre-processing - is, in our experience, the largest single reliability win available.

The **report and finance generators** hold zero tools. The report agent receives metrics computed and frozen by code (conversion significance, revenue attribution, fees) and is instructed to narrate only the numbers given; it cannot compute, so it cannot miscompute. The finance assistant emits a structured draft invoice for human review and has no capability to issue or send anything.

The pattern across all six: tool count is inversely proportional to blast radius. Where an error would be expensive, the agent's action space approaches zero and the surrounding system does the work.

5. Human-in-the-Loop Placement

The question is not whether humans review agent output but where the review gate sits. Three placements recur in our deployment and in the broader practice:

Draft-only boundaries. Where an agent's output crosses into an external system, it lands as a draft. Our email agent designs a lifecycle flow as a structured content model, a deterministic renderer produces the ESP-safe HTML, and the integration pushes the result into the client's Klaviyo account strictly as drafts - the code path to send or schedule does not exist. A human activates the flow in the ESP itself. Capability absence is a stronger guarantee than capability restraint.

Approval steps between agents. In our experimentation pipeline the design agent's mockup is approved by a human before the code agent runs, and the code agent's output is approved before it is attached to the task. The gates sit exactly where an error would propagate.

Free interior, gated edges. Within the loop, read operations run unsupervised and low-stakes writes (a status change, a comment) run with audit-trail-after rather than approval-before. Reserving human attention for the expensive

edges keeps review meaningful; gating everything trains reviewers to rubber-stamp.

6. Failure Modes: Hallucinated Action, Context Rot, and Injection

Hallucinated actions. Hallucination in generation - fluent, confident, unfaithful output - is thoroughly documented.[10] In a tool-using agent the failure mutates: the model does not merely state a wrong fact, it acts on one - updating the wrong record, inventing a figure in a client report. The mitigations that work are structural, not rhetorical: typed tools whose arguments must resolve against real state, validators that reject what does not exist, and the zero-tool pattern of Section 4 for numeric content. Prompting an agent to 'be careful' is not a mitigation.

Context degradation. Long agent sessions accumulate observations, and retrieval quality falls with position in the window.[9] Symptoms include re-executing completed steps and ignoring early constraints. Mitigations: session-scoped agents (a fresh, staged context per task rather than one long-lived conversation), aggressive truncation of tool results, and artifacts on disk as the durable memory instead of the chat transcript.

Indirect prompt injection. The defining security problem of tool-using agents: any text the agent reads is a potential instruction channel. Greshake et al. demonstrated that adversarial instructions embedded in retrieved content - a web page, a document, an email - can hijack an LLM-integrated application without the user ever typing anything malicious.[11] The OWASP LLM Top 10 now ranks prompt injection as the leading risk class for LLM applications.[12] Production consequences for agent design: (i) fence untrusted inputs - our agents receive client briefs and scraped page content wrapped in explicit data delimiters with instructions that the content is never a command, and fence-breakout sequences are neutralized before the prompt is assembled; (ii) assume fencing fails, and rely on least privilege - an injected instruction cannot exfiltrate files from an agent that has no filesystem, or send email from an

integration that can only create drafts; (iii) keep authorization below the model (Section 4), so a hijacked agent still cannot exceed its user's permissions. Injection resistance is, again, an architecture property.

7. Evaluation and Operations

Benchmarks establish that capability exists; they do not establish that a deployment is sound. SWE-bench demonstrated how far agentic capability has come on real repository issues, and also how much headroom remains.[13] Kapoor et al. argue that agent benchmarks systematically overstate practical readiness by ignoring cost, reproducibility, and the gap between maximal-accuracy configurations and anything an operator would actually run.[14] Both cautions transfer directly: an agent that resolves a benchmark task at any token cost is not evidence that the same loop is economical at production volume.

Using a model to grade agent output scales review, and agreement with human raters is high in benchmark settings[15], but judge models import their own biases (position, verbosity, self-preference) and should be treated as a triage layer above, not a replacement for, human gates at expensive boundaries.

Operationally, the practices that mattered most in our deployment are mundane: every agent run wrapped in failure classification and alerting; systemic failures (authentication, provider availability) distinguished from task-level ones and paged to an operator with deduplication; a daily credential probe so a dead token surfaces within a day rather than at a client deadline; and provider independence through a single configuration switch between two model vendors, which converts a provider incident from an outage into a config change.

8. Practical Guidance

For teams putting tool-using agents into a real business, the review above compresses to seven rules.

1. Start from the workflow, not the model: identify the repetitive task, then give an agent the minimum tools that task requires.[8]
2. Enforce authorization below the model, at the data layer, in code paths shared with human users.
3. Move reliability-critical work (scraping, rendering, arithmetic, validation) into deterministic code; reserve the agent for judgment and language.
4. Give numeric and financial content the zero-tool treatment: code computes, the model narrates.
5. Make outputs that cross system boundaries draft-only, and put explicit approval gates between pipeline stages.
6. Treat every input the agent reads as untrusted; fence it, and design so that a successful injection still has nowhere to go.[11][12]
7. Evaluate with cost on the scoreboard and instrument failures from day one.[14]

9. Conclusion

The research trajectory from ReAct through function calling settled the question of whether language models can use tools. The question that decides production outcomes is different: what tools, under whose authority, validated where, reviewed by whom. Across six agents in daily operation, the systems that earned trust share an unglamorous profile - narrow jobs, small typed tool surfaces, authorization below the model, deterministic code around it, and humans at the expensive edges. Agent reliability is an architecture property before it is a model property, and it is available to any team willing to design for it.

References

1. Yao, S., Zhao, J., Yu, D., et al. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. *ICLR 2023*. https://doi.org/10.48550/arXiv.2210.03629

2. Schick, T., Dwivedi-Yu, J., Dessì, R., et al. (2023). Toolformer: Language Models Can Teach Themselves to Use Tools. *NeurIPS 2023*. https://doi.org/10.48550/arXiv.2302.04761

3. Qin, Y., Liang, S., Ye, Y., et al. (2023). ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. *arXiv preprint*. https://doi.org/10.48550/arXiv.2307.16789

4. Wang, L., Ma, C., Feng, X., et al. (2024). A Survey on Large Language Model based Autonomous Agents. *Frontiers of Computer Science*. https://doi.org/10.1007/s11704-024-40231-1

5. Mialon, G., Dessì, R., Lomeli, M., et al. (2023). Augmented Language Models: a Survey. *arXiv preprint*. https://doi.org/10.48550/arXiv.2302.07842

6. Patil, S. G., Zhang, T., Wang, X., et al. (2023). Gorilla: Large Language Model Connected with Massive APIs. *arXiv preprint*. https://doi.org/10.48550/arXiv.2305.15334

7. Shinn, N., Cassano, F., Berman, E., et al. (2023). Reflexion: Language Agents with Verbal Reinforcement Learning. *NeurIPS 2023*. https://doi.org/10.48550/arXiv.2303.11366

8. Schluntz, E., & Zhang, B. (2024). Building Effective AI Agents. *Anthropic*. https://www.anthropic.com/engineering/building-effective-agents

9. Liu, N. F., Lin, K., Hewitt, J., et al. (2024). Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics (TACL)*. https://doi.org/10.48550/arXiv.2307.03172

10. Ji, Z., Lee, N., Frieske, R., et al. (2023). Survey of Hallucination in Natural Language Generation. *ACM Computing Surveys*. https://doi.org/10.1145/3571730

11. Greshake, K., Abdelnabi, S., Mishra, S., et al. (2023). Not what you've signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. *arXiv preprint*. https://doi.org/10.48550/arXiv.2302.12173

12. OWASP Foundation (2025). OWASP Top 10 for Large Language Model Applications. *OWASP Foundation*. https://owasp.org/www-project-top-10-for-large-language-model-applications/

13. Jimenez, C. E., Yang, J., Wettig, A., et al. (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. *ICLR 2024*. https://doi.org/10.48550/arXiv.2310.06770

14. Kapoor, S., Stroebl, B., Siegel, Z. S., et al. (2024). AI Agents That Matter. *arXiv preprint*. https://doi.org/10.48550/arXiv.2407.01502

15. Zheng, L., Chiang, W.-L., Sheng, Y., et al. (2023). Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. *NeurIPS 2023 Datasets and Benchmarks Track*.

https://doi.org/10.48550/arXiv.2306.05685

Cite this

APA Copy

```
Bhaskar Roy Sarkar (2026). Tool-Using LLM Agents in Production
Business Workflows: A Practitioner Review. cazyweb Research.
https://cazyweb.com/research/tool-using-llm-agents-production-
workflows
```

BIBTEX Copy

```
@techreport{tool-using-llm-agents-production-workflows,
  author = {Bhaskar Roy Sarkar},
  title = {Tool-Using LLM Agents in Production Business Workflows: A
Practitioner Review},
  institution = {cazyweb Research},
  year = {2026},
  url = {https://cazyweb.com/research/tool-using-llm-agents-
production-workflows}
}
```

RELATED READING

How we run our agency on AI agents

Should your business build a custom AI agent?

Want a team to run experiments like this for you? See how we help

Let's build something that converts.

Start a project with us, or join the bench of specialists and agencies we deliver with.

Start a project ↗

cazyweb

The engineering and growth team behind leading brands and companies across the US and Europe.

CRO and ecommerce insights

Occasional, genuinely useful notes. No fluff.

you@company.com

Subscribe →

Work email only. No spam. Unsubscribe any time. Privacy.

Solutions

- Product Optimization
- Experimentation Program
- cazyweb Platform

For Ecommerce

- CRO Program
- Email & Retention
- Paid Ads
- Landing Pages
- Revenue Calculator FREE
- A/B Test Calculator FREE
- CRO Agency
- Shopify CRO
- Ecommerce Growth
- Website Performance

Company

- About
- For Brands
- For Agencies
- Careers HIRING
- Contact

Explore

- Services
- Pricing
- Performance program
- Subscription
- Website + CRO from $99
- Case Studies
- Blog
- CRO Ideas
- Apps
- Learn

CazyWeb IT Solutions
GSTIN: 19JTLPS8476P2ZZ
hello@cazyweb.com

© 2026 cazyweb. All rights reserved.

Privacy Policy Terms of Service Cancellation & Refunds Cookie Settings