Deploying Programmatic Tool Calling with Pre-Execution Validation for Production Agentic Systems

Nhan Nguyen
Amazon Web Services
ntrnguye@amazon.com

Hadi Habibzadeh
Amazon Web Services
hhabibz@amazon.com

Ahmad Magdy Eid
Amazon Web Services
magdyeid@amazon.com

Abstract

Production tool calling for agentic systems must satisfy four operational requirements: lower per-invocation cost, low latency, adaptability to evolving tool catalogs, and the ability to catch errors before execution. The ReAct paradigm keeps the large language model (LLM) in the loop, but at production scale, reprocessing intermediate results inflates cost and latency and degrades answer quality. Programmatic tool calling (PTC) offers an alternative; the LLM generates a single script that resolves dependencies and extracts relevant fields in one inference pass. PTC is a harder generation task, however, because the LLM must produce code that handles complex, variable responses without observing intermediate results, and existing remedies each fail at least one of the four requirements. We describe a production deployment of PTC serving 26,528 invocations, paired with a three-layer abstract syntax tree (AST)-based validator that rejects invalid scripts before execution and requires no training data, preserving adaptability to evolving tool catalogs. This reduced average input tokens by 77%, latency by 21%, and increased the answered-query rate by 5.6 percentage points across 19 production agents.

1 Introduction

Agentic systems use tool calling (i.e., the mechanism by which agents obtain query context from the environment) to perform real-world tasks. In production, tool calling faces four operational requirements: (1) low per-invocation cost at scale, (2) low latency under varying query complexity, (3) adaptability to evolving tool catalogs without expensive retraining, and (4) the ability to catch errors before execution because incorrect tool calls can produce side effects that cannot be reversed. These constraints, empirically confirmed across diverse production deployments (Pan et al., 2025), narrow the design space.

Tool calling requires both *dependency resolution* (i.e., passing one tool's output as another's input) and *selective result extraction* (i.e., identifying and returning only the relevant portions of tool responses), both of which vary with the query and with intermediate results. The difficulty of dependency resolution and selective result extraction grows with the number and complexity of available tools (Qin et al., 2024); production workflows may require thousands of heterogeneous application programming interfaces (APIs) that have deeply nested response structures.

One common approach to enable tool calling in agentic workflows is to use *ReAct* (Yao et al., 2022). This approach uses an LLM-in-the-loop framework consisting of multiple tool-call iterations, where the LLM observes each tool response, extracts relevant parameters for subsequent calls, and reasons about which portions of the results matter. This iterative approach works well when the tool set is small and each query requires few calls. In production environments, however, queries often require a large tool set (thousands of tools) and many tool calls (tens of calls per query) (Tang et al., 2025). At this scale, ReAct inflates cost and latency; intermediate results re-enter the LLM's context on every iteration, are re-tokenized, and accumulate as noise that also lowers answer quality (Liu et al., 2026).

Programmatic tool calling (PTC) addresses the cost and latency requirements (and the associated noise that lowers answer quality) by replacing the iterative tool-calling loop with a single code-generation pass. The LLM generates a script that handles dependency resolution and selective result extraction programmatically; the script executes in a sandbox without further LLM involvement, returning only the extracted results to the agent. Intermediate results are processed within the sandbox and do not re-enter the LLM's context. Thus, PTC's per-invocation input tokens and LLM infer-

ence latency do not grow with the number of tool calls, unlike ReAct where both grow as intermediate results accumulate in context across turns.

Code-based tool calling has been explored in the literature. CodeAct (Wang et al., 2024) and TaskWeaver (Qiao et al., 2023) demonstrate that code actions outperform JSON-based tool calling. ViperGPT (Surís et al., 2023) applies code generation to compose vision-and-language models, and subsequent work (Zhang et al., 2025; Gao et al., 2025) reports further token-efficiency and parallel-execution gains. These systems evaluate on offline benchmarks (Liu et al., 2024; Yang et al., 2023) with small, fixed tool sets. Production deployment over large, evolving tool catalogs with real users remains underexplored; prior enterprise work (Osuagwu et al., 2025) focuses on tool retrieval rather than programmatic invocation.

PTC introduces a new failure mode that conflicts with the reliability requirement. The LLM must produce a script that accesses the correct fields in each tool's response and uses correct identifiers; fields are often deeply nested and may or may not be present depending on the environment, and ungrounded identifiers can cause unintended data access or rate-limit pressure when the wrong customer's resources are queried. Unlike ReAct, where the LLM can observe and correct a mistake at each iteration, a PTC script executes without intermediate checkpoints. Thus, errors propagate silently. Internal testing with deliberately incorrect input values (specifically, geographic region identifiers) showed silent value fabrication in 3.4% of cases, where the LLM substituted the user-supplied identifier with a plausible but incorrect alternative. Prompt-based rules proved insufficient for preventing these failures; we observed cases where the LLM correctly stated a constraint in its reasoning trace and then violated it in the generated code.

Existing mechanisms for LLM-code correctness each address part of the problem but fail at least one operational requirement. Multi-turn refinement (Wang et al., 2024) retries on execution errors, conflicting with latency. Constrained decoding (Dong et al., 2025) enforces output format at generation time but not semantic content; the LLM can produce a structurally valid script that still contains fabricated values. Sandboxing prevents damage from incorrect execution but does not prevent the code from being wrong. Benchmarks and retrieval-augmented grounding (Patil et al., 2025, 2024) detect API hallucination offline but do not validate production scripts at runtime. Broader tool-augmented LLM research (Schick et al., 2023; Xu et al., 2025; Jiayang et al., 2026) addresses the problem at the model-training level, conflicting with operational simplicity and adaptability. A lightweight pre-execution validation approach (one that requires no training data and no post-execution recovery) satisfies all four requirements simultaneously.

We present the design and production evaluation of a system that adopts query-time PTC as the primary execution path in an enterprise agentic platform. Our contributions are threefold. (1) We describe the end-to-end system design that makes PTC practical at scale, including query-conditioned tool discovery to keep prompts small, a single-pass code generation step, and a restricted execution sandbox with bounded parallel execution. (2) We propose a three-layer abstract syntax tree (AST)-based validator that inspects every generated script before execution and rejects code containing unsafe constructs, non-existent API methods, or parameter values absent from the user's input. (3) We provide production evidence from 26,528 invocations across 19 agents demonstrating that PTC reduced average input tokens by 77%, latency by 21%, and increased the answered-query rate by 5.6 percentage points; for value fabrication specifically, AST-level validation outperformed a prompt-based rule by nearly $2\times$ in a controlled comparison ($+10.2$ pp vs. $+5.9$ pp).

The rest of this paper is organized as follows. Section 2 describes the system architecture; Section 3, the methodology; Section 4, the production results; and Sections 5 and 6 conclude and discuss limitations.

2 System Architecture

Our system adopts PTC as the primary execution path for tool calling in an agentic workflow. Given a user query and, optionally, a set of agent-declared tool identifiers (i.e., fully qualified API method names), PTC generates and executes a single validated script. The script batches all calls whose parameters can be resolved ahead of time, handling both dependency resolution and selective result extraction. A short residual ReAct loop may follow when subsequent calls depend on the content of prior tool responses; this loop runs on the legacy iterative path.

Figure 1 illustrates the end-to-end system, con-

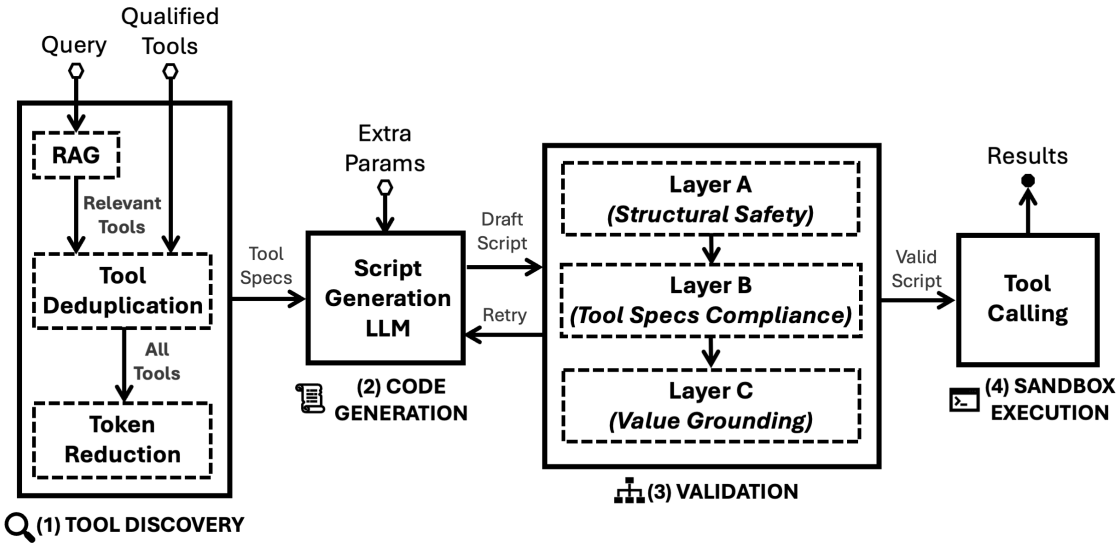

Figure 1: The PTC architecture, consisting of four components: (1) *tool discovery* retrieves specifications for the requested tools; (2) *code generation* produces a script that calls the tools, handling dependency resolution and selective result extraction; (3) *validation* inspects the generated script for errors before execution; and (4) *sandbox execution* runs the validated script in a restricted environment and returns the results.

sisting of four components: (1) *tool discovery* retrieves the relevant tool specifications from the catalog given the query and tool identifiers; (2) *code generation* produces a script that calls the requested tools with the correct parameter passing and result extraction logic; (3) *validation* inspects the generated script and rejects code containing errors before execution; and (4) *sandbox execution* runs the validated script in a restricted environment and returns the results. The rest of this section describes each component in detail.

2.1 Tool Discovery

Tool discovery identifies the relevant tools for a given agent's request and retrieves their specifications (i.e., structured documents describing each tool's method name, input parameters, and response schema). The agent may pass explicit tool identifiers in its request (when it knows the required tools), or it may pass only the query itself (when it does not know the required tools) and rely on the system to discover relevant tools from the catalog.

The tool discovery component uses two retrieval mechanisms that run concurrently: (1) semantic search over the tool catalog using an embedding index, returning the top-K specifications by cosine similarity to the query; and (2) exact-match lookup, which retrieves the specifications for any tool identifiers the agent explicitly provides. The component deduplicates and merges the retrieved specifications, capping the merged list at N to bound prompt size. The selected specifications are then serialized using a token-efficient notation that removes redundant structural characters present in standard JSON (e.g., repeated quotation marks, braces, and delimiters), reducing their size without altering their information content.

Tool discovery caches results at two levels; one cache keyed by the normalized query string and one keyed by the sorted tool identifier list. Cached results allow the component to bypass retrieval entirely for repeated queries, eliminating retrieval latency on cache hits.

2.2 Code Generation

The code generation component takes the user query, a set of extracted parameters (e.g., region or account identifier), and the tool specifications retrieved by tool discovery to generate the PTC script. The component makes a single LLM call that returns a JSON object containing a reasoning plan and the generated script. The output structure is enforced via the inference API's constrained-decoding mode (Zeng and Evans, 2026). The generated script is a Python function that uses a system-provided call primitive to invoke tools, resolves dependencies by passing parameters from one tool's response to the next, and performs selective result extraction by filtering and returning only the relevant portions of each response. The system prompt supplies code conventions, executable examples of common patterns (parallel-independent

calls, sequential-dependent calls, and parameter-chaining calls), and a list of common failure patterns drawn from production errors. The system prompt is static across all requests, enabling server-side prompt caching to reduce latency and cost on every invocation.

2.3 Validation

The validation component inspects every generated script before execution and rejects code that contains errors the LLM introduced during generation. The validator parses the generated script into an AST and applies three layers of checks, each targeting a distinct class of error:

Layer A – Structural safety. This layer walks the AST and rejects code that uses constructs outside the permitted runtime including disallowed imports, execution built-ins (e.g., `eval`, `exec`), file or network I/O, and any access pattern that bypasses the system-provided call primitive.

Layer B – Tool specification compliance. This layer resolves every call primitive invocation in the AST and verifies that the tool identifier, method name, and required parameters match the tool specifications retrieved during discovery. A script that calls a non-existent method or omits a required parameter is rejected at this stage.

Layer C – Value grounding. This layer parses the script's AST and traces every string literal used as an API parameter back to its source through variable bindings. A value is accepted if it matches tokens from the user query (with identifier-aware tokenization that splits structured resource identifiers into their component fields), caller-supplied resource fields, tool-spec identifiers, or pre-filled values; equivalence tables resolve known aliases (e.g., a resource's human-readable display name and its internal identifier) before matching. Any literal that cannot be traced to such a source is treated as fabricated and rejected.

Layer C prevents the failure class described in Section 1; the LLM substituting a plausible but incorrect value (e.g., replacing a user-supplied identifier with a more common alternative) that would otherwise execute successfully against unintended resources. A validation failure triggers a single retry in which the full error list is appended to the prompt. The first retry recovers most validation failures. Additional retries beyond the first rarely improve recovery; thus, persistent failures are surfaced to the agent, which can then request clarification from the user.

2.4 Sandbox Execution

The sandbox execution component runs validated scripts in a restricted Python environment built fresh for each request. The environment exposes only the system-provided call primitives and a small set of allow-listed standard library modules; all other imports, file system access, and network operations are unavailable. A call primitive sends a single tool invocation to the appropriate backend and performs response normalization (e.g., resolving default parameters, stripping SDK metadata, and standardizing timestamp formats) so that generated scripts can operate on a consistent response structure regardless of the underlying tool. A parallel-calls variant executes multiple independent tool invocations concurrently via a bounded thread pool, reducing end-to-end latency when the script issues several independent calls. Errors from tool invocations are returned as values rather than raised as exceptions, allowing the generated script to handle them with ordinary control flow. An execution timeout interrupts the sandbox after a fixed duration and returns any partial results collected up to that point. The sandbox is intentionally lightweight compared to full process-level isolation approaches (e.g., container-based sandboxes) because the validation layer has already constrained the code surface to only call primitives and in-memory data transformations, eliminating the need for heavier isolation mechanisms.

3 Method

This section describes the deployment context, evaluation setup, metrics, and validator comparison methodology used to assess the impact of adopting PTC in production.

3.1 Deployment Setup

The system is deployed within a cloud support platform where agents diagnose customer issues by calling cloud service APIs on the customer's behalf. At the time of measurement, the platform hosted over 1,600 agents in production, collectively serving six-figure daily invocation volumes. Each agent is configured with a standard operating procedure that specifies which tools to call and how to reason over their outputs for a given class of customer query. The tool catalog contains over 13,000 API specifications spanning hundreds of cloud services. Individual tool specifications describe complex parameter schemas and return deeply nested

response structures that vary depending on the customer's resource configuration. A typical diagnostic query requires the agent to call between two and ten tools, where each subsequent call depends on values extracted from prior responses. Queries are customer-initiated and concern specific resources (e.g., a particular load balancer, database instance, or network configuration), meaning the correct tool parameters are unique to each query and cannot be inferred from training data alone.

All code generation uses Claude Haiku 4.5 (Anthropic, 2025) accessed via a hosted inference API. Tool discovery returns the top-K specifications by semantic similarity to the query (with $K = 50$), merged with exact-match lookups and capped at $N = 20$ specifications per request; tool specifications are serialized in TOON (TOON contributors, 2024), a token-efficient notation that omits redundant structural characters present in JSON. The sandbox enforces an execution timeout of 60 s. The majority of production invocations pass the agent-declared tool list; query-only invocations are a minority path.

3.2 Evaluation Setup

We compare the platform's behavior across two non-overlapping production windows: a pre-PTC window (10 days, 95,155 invocations) running the legacy ReAct-style iterative tool loop, and a post-PTC window (8 days, 26,528 invocations) after PTC integration. Both windows serve the same agent population under normal production traffic; no traffic was artificially routed or filtered. The pre-PTC window contains a higher volume of internal validation traffic, generated by engineers exercising the agent platform during its earlier rollout phase. This traffic follows the same code path and metric definitions as customer traffic, and testers had no visibility into the underlying tool-calling strategy; per-invocation aggregates therefore remain comparable across windows. The two windows are separated by approximately four weeks, during which PTC was iteratively developed and stabilized; the agent population and tool catalog remained substantively stable across the gap.

We focus the evaluation on the top 20 agents by invocation count in the post-PTC window (each with at least 340 invocations). Low-volume agents produce point estimates with wide confidence intervals that are difficult to interpret; restricting to high-volume agents ensures meaningful per-agent comparisons. Of the 20, 19 also appeared in the pre-PTC window with sufficient volume for comparison; the remaining agent did not exist in the pre-PTC period.

3.3 Metrics

We evaluate PTC along three dimensions: cost, latency, and answer quality.

Cost. We measure cost as input tokens per invocation, which is the primary driver of LLM inference cost; per-token pricing scales with the input context, while output tokens are comparatively small and stable across conditions.

Latency. We measure latency as LLM inference time per invocation, excluding tool execution time, since it depends primarily on backend service response times.

Answer quality. We measure answer quality as the answered-query rate, i.e., the proportion of invocations that produce a substantive answer to the customer's question. Every invocation terminates in exactly one of three outcomes: `Answered` (a substantive reply based on tool results), `NoAnswer` (the agent does not produce a substantive answer, due to either an internal failure or an out-of-scope determination made before or after tool execution), or `MissingResources` (the agent requests clarification because the query lacks required parameters). A higher answered-query rate indicates that fewer invocations are lost to non-answers (internal failures or unnecessary out-of-scope classifications) or avoidable clarification requests.

3.4 Validator Comparison

We compare two interventions targeting silent value fabrication, the failure class described in Section 1. The first is a prompt rule instructing the LLM to preserve user-supplied values. The second is the Layer C value-grounding validator, which mechanically rejects ungrounded literals. Each was shipped as a single release and measured against the same traffic, model, and workload in a 24–48 hour window flanking the deployment. We measure `SuccessRate`, the proportion of invocations whose generated script passed validation and executed without error.

The two releases occurred at different points in the system's evolution, so absolute `SuccessRate` values reflect the system state at each release rather than a chained sequence. The comparison is observational rather than randomized; the single-variable structure (one intervention per release) and short

measurement windows make confounding from traffic or query-distribution shifts unlikely.

4 Results

	Input tokens	Latency
Pre-PTC	7,409	35.1 s
Post-PTC (Δ)	1,686 (−77.2%)	27.7 s (−21.1%)

Table 1: Cost and latency metrics per invocation, averaged across 19 paired agents.

Table 1 summarizes cost and latency metrics across the pre-PTC window (95,155 agent invocations) and post-PTC window (26,528 agent invocations), aggregated over all 19 paired agents. Average input tokens per invocation dropped from 7,409 to 1,686 (−77.2%), and inference latency dropped from 35.1 s to 27.7 s (−21.1%). Weighting each of the 19 agents equally, the mean per-agent input-token reduction is 9.4% (median 15.4%). 16 of 19 agents reduced input tokens, while 3 increased.

Outcome	Pre-PTC	Post-PTC	Δ
Answered	28.2%	33.8%	+5.6 pp
NoAnswer	48.9%	40.0%	−8.9 pp
MissingResources	22.9%	26.2%	+3.4 pp

Table 2: Response-outcome distribution across 19 paired agents (per-agent average).

Table 2 reports the response-outcome distribution across the same 19 paired agents. The answered-query rate increased from 28.2% to 33.8% (+5.6 pp), `NoAnswer` decreased from 48.9% to 40.0% (−8.9 pp), and `MissingResources` increased from 22.9% to 26.2% (+3.4 pp). On the highest-traffic agent (66% of post-PTC volume), the answered-query rate improved from 89.7% to 99.1% (+9.4 pp).

Table 3 compares the Layer C value-grounding validator against a prompt-only baseline addressing the same failure class. The validator delivered +10.2 pp over a 24-hour measurement window; the prompt rule delivered +5.9 pp over a 48-hour window. Across the broader 12-day rollout, additional refinements lifted `SuccessRate` from 72.5% to 98.7%.

Change	SuccessRate	Δ
Prompt rule	72.5% → 78.5%	+5.9 pp
Layer C validator	76.1% → 86.3%	+10.2 pp

Table 3: Comparison of prompt-based and validator-based approaches to the same failure class. Absolute `SuccessRate` values reflect the system state at each release.

5 Conclusion

We presented a system that adopts programmatic tool calling (PTC) as the primary execution path for tool calling in a production agentic platform. PTC generates a single script that handles dependency resolution and selective result extraction programmatically, eliminating the per-iteration cost of re-processing intermediate results through the LLM. A three-layer AST-based validator rejects scripts containing unsafe constructs, non-existent API methods, or fabricated parameter values before execution. Measured across 19 production agents, PTC reduced input tokens per invocation by 77% and latency by 21% on average, and increased the answered-query rate by 5.6 percentage points. The combined system satisfies all four operational requirements identified in Section 1: low per-invocation cost, low latency, adaptability to evolving tool catalogs, and the ability to catch errors before execution.

Our comparison shows that prompt-based instructions does not prevent value fabrication in generated code; the model obeys constraints in its reasoning trace while violating them in the script it produces. The AST-level validator outperformed the prompt-based rule by nearly 2× (+10.2 pp vs. +5.9 pp), mechanically enforcing constraints on the code itself rather than asking the model to do so. This suggests that teams deploying code-generation agents over large API surfaces should invest in structural validation before iterating on prompts.

PTC reduces input tokens not by generating less useful content but by reallocating the inference budget. The system spends the budget once, up front, rather than on every iteration. The increase in `MissingResources` responses (+3.4 pp) reflects cases where the legacy system would have executed against a fabricated identifier. PTC instead surfaces a clarification request, a trade-off we consider correct for a support deployment where ungrounded answers carry real cost.

6 Limitations

PTC generates a single script per invocation and executes it without further LLM interaction. This

means the system cannot incorporate branching logic that depends on a tool response value discovered at runtime—for example, selecting a remediation path based on the content of an error message. Such queries fall back to the iterative execution path. For queries requiring exactly one API call, PTC also introduces net overhead (code generation, validation, and sandbox setup) that exceeds the cost of a direct tool call.

All results are produced by a single frontier LLM, and we have not evaluated PTC quality on other models. Some implementation decisions are coupled to the specific inference API used, making portability to other LLMs an open question. The post-PTC evaluation window is 8 days; whether quality gains hold across shifting query distributions, model updates, or tool-catalog expansions is not established by this study.

We report aggregate deltas across 19 paired agents but do not compute per-agent confidence intervals or significance tests. The aggregate numbers across 19 paired agents (approximately 121,000 total invocations) are robust, but individual agent-level changes carry wide confidence intervals, particularly on low-volume agents. Finally, the value-grounding validator accepts string values only from a closed set of allowed sources (user query, caller-supplied resources, tool specifications, and pre-filled values). Values that the model correctly recalls from training (such as API enum constants or service-specific status codes) are rejected unless they appear in one of these sources. This deliberate precision-over-recall trade contributes to the observed increase in `MissingResources` responses, though we consider false rejections preferable to silent execution against fabricated identifiers.

References

PBC Anthropic. 2025. Introducing claude haiku 4.5. Accessed: 2026-04-15.

Yixin Dong, Charlie F Ruan, Yaxing Cai, Ziyi Xu, Yilong Zhao, Ruihang Lai, and Tianqi Chen. 2025. XGrammar: Flexible and efficient structured generation engine for large language models. *Proceedings of Machine Learning and Systems*, 7.

Silin Gao, Jane Dwivedi-Yu, Ping Yu, Xiaoqing Ellen Tan, Ramakanth Pasunuru, Olga Golovneva, Koustuv Sinha, Asli Celikyilmaz, Antoine Bosselut, and Tianlu Wang. 2025. Efficient tool use with chain-of-abstraction reasoning. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 2727–2743.

Cheng Jiayang, Xin Liu, Zhihan Zhang, Haoyang Wen, Zixuan Zhang, Qingyu Yin, Shiyang Li, Priyanka Nigam, Bing Yin, Chao Zhang, and 1 others. 2026. Training llms for multi-step tool orchestration with constrained data synthesis and graduated rewards. *arXiv preprint arXiv:2603.24709*.

Boyan Liu, Gongming Zhao, and Hongli Xu. 2026. Utility-guided agent orchestration for efficient llm tool use. *arXiv preprint arXiv:2603.19896*.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, and 1 others. 2024. AgentBench: Evaluating LLMs as agents. In *International Conference on Learning Representations*, volume 2024, pages 52989–53046.

Richard Osuagwu, Thomas Cook, Maraim Masoud, Koustav Ghosal, and Riccardo Mattivi. 2025. ScaleCall–agentic tool calling at scale for Fintech: Challenges, methods, and deployment insights. *arXiv preprint arXiv:2511.00074*.

Melissa Z Pan, Negar Arabzadeh, Riccardo Cogo, Yuxuan Zhu, Alexander Xiong, Lakshya A Agrawal, Huanzhi Mao, Emma Shen, Sid Pallerla, Liana Patel, and 1 others. 2025. Measuring agents in production. *arXiv preprint arXiv:2512.04123*.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. 2025. The Berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2024. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37:126544–126565.

Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, and 1 others. 2023. TaskWeaver: A code-first agent framework. *arXiv preprint arXiv:2311.17541*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, and 1 others. 2024. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *International Conference on Learning Representations*, volume 2024, pages 9695–9717.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551.

Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. ViperGPT: Visual inference via python execution for reasoning. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 11888–11898.

Yihong Tang, Kehai Chen, Liang Yue, Jinxin Fan, Caishen Zhou, Xiaoguang Li, Yuyang Zhang, Mingming Zhao, Shixiong Kai, Kaiyang Guo, and 1 others. 2025. Empowering real-world: A survey on the technology, practice, and evaluation of llm-driven industry agents. *arXiv preprint arXiv:2510.17491*.

TOON contributors. 2024. Toon: Token-oriented object notation.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*.

Hongshen Xu, Zihan Wang, Zichen Zhu, Lei Pan, Xingyu Chen, Shuai Fan, Lu Chen, and Kai Yu. 2025. Alignment for efficient tool calling of large language models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 17787–17803.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2023. InterCode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems*, 36:23826–23854.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. ReAct: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Jeffrey Zeng and Jonathan Evans. 2026. Structured outputs on Amazon Bedrock: Schema-compliant AI responses. https://aws.amazon.com/blogs/machine-learning/structured-outputs-on-amazon-bedrock-schema-compliant-ai-responses/.

Cedegao E Zhang, Cédric Colas, Gabriel Poesia, Joshua B Tenenbaum, and Jacob Andreas. 2025. Code-enabled language models can outperform reasoning models on diverse tasks. *arXiv preprint arXiv:2510.20909*.