Scaling Enterprise Cloud Agents to Thousands of APIs: An Empirical Study of Agent Architecture and MCP Tool Boundaries

Sandip Ghoshal*, **Khalil Mrini***, **Maram Assi**, **Yadan Fan**, **Jyotika Singh**, **Miguel Ballesteros**, **Weiyi Sun**, **Anshul Mittal**, **Yassine Benajiba**, **Sujeeth Bharadwaj**, **Sujith Ravi**, **Dan Roth**
Oracle AI
Correspondence: sandip.ghoshal@oracle.com; khalil.mrini@oracle.com

Abstract

Enterprise cloud agents increasingly operate over thousands of APIs and SDK operations. Tool use therefore becomes a large-scale classification and grounding problem over API operations, not a simple function-calling task. Earlier production-derived architectures either expose broad operation catalogs to the model or hide selection inside planner agents, causing excessive context growth, repeated discovery calls, and preventable invocation errors. We propose the *Operation-Visible Agent Architecture*, which keeps operation search, schema inspection, validation feedback, and API invocation visible to a centralized ReAct loop through a deterministic Operation-Resolver MCP. We compare this architecture with lead-worker planning and ReAct-based execution variants to assess how agent orchestration and MCP tool-boundary design affect enterprise-agent performance. On internal cloud-management benchmark tasks spanning 280+ SDK clients and ∼8K operations, the proposed architecture improves full-success rate from 68% to 77% over the production-derived baseline, while reducing total token usage by 54.1%, median latency by 43.3%, LLM calls by 37.8%, and tool errors by 52.4%. The main lesson is that, at enterprise API scale, tool-boundary design is not an implementation detail: parallel tool calling improves efficiency, but task success improves most when operation resolution remains visible to the main reasoning loop.

1 Introduction

Language-model (LM) systems are increasingly moving from text generation to action-oriented assistance: interpreting user intent, retrieving context, and executing actions through tools, databases, and APIs. Prior work on augmented LMs, RAG, ReAct, and tool-use benchmarks shows the value of connecting models to external knowledge and computation (Mialon et al., 2023; Lewis et al., 2020; Yao et al., 2023; Schick et al., 2023; Li et al., 2023; Qin et al., 2024; Patil et al., 2024). Enterprise cloud agents operate over large SDK surfaces rather than small fixed tool sets. A *client* is a service-specific SDK interface; an *operation* is an executable client method with schema, authorization constraints, and execution metadata. Selecting an operation requires choosing the right client, method, and grounded parameters.

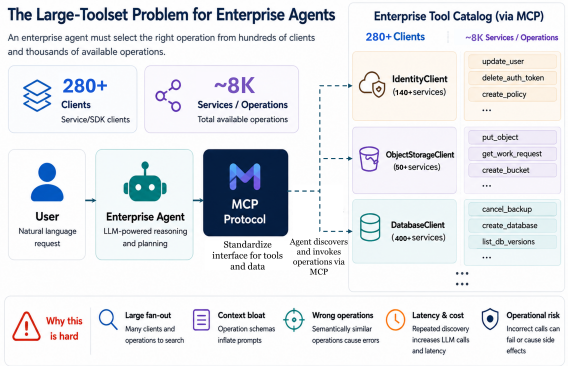

Figure 1: Large-toolset problem for enterprise agents: mapping a user request through the agent and MCP interface to the correct client, operation, and parameters across hundreds of clients and thousands of operations, creating tool-selection, context-size, latency, and reliability challenges.

We study a production-deployed large enterprise cloud agent that invokes cloud APIs through a Model Context Protocol (MCP) interface over 280+ SDK clients and ∼8K exposed operations. Here, tool use is both a retrieval problem and an architecture-boundary problem: the agent must identify the right operation and parameters without flooding model context with irrelevant SDK structure. Poor grounding causes broad catalog scans, guessed operations or parameters, unnecessary clarification requests, higher latency and cost, context noise, and harder-to-debug failures. These con-

*These authors contributed equally to this work.

EMNLP 2026 Industry Track, pages 3040–3055
October 24-29, 2026 ©2026 Association for Computational Linguistics

straints are not merely benchmark artifacts; they directly affect production cost, latency, debugging, and reliability for enterprise assistants operating over evolving API catalogs.

After observing two earlier designs, a generic invocation architecture with frequent operation and parameter errors, and a planner-based MCP architecture (section 4) that improved selection but added LLM calls and duplicated context, we propose the *Operation-Visible Agent Architecture*, shown in Figure 2. The *Operation-Visible Agent Architecture* pairs ReAct Parallel Tool Calling with the deterministic Operation-Resolver MCP. The resolver exposes three explicit steps: searching for candidate operations, describing their schemas and constraints, and validating/executing API calls. Because these steps remain in the main ReAct trajectory, the agent can inspect candidates, use validation feedback, observe API results, and produce the final response without delegating operation selection to a hidden planner or specialist subagent. ReAct Parallel Tool Calling further allows one reasoning step to issue multiple independent API calls when their arguments are known, reducing model round trips. Our contributions are:

- **Problem.** We frame enterprise cloud-agent tool use as *large-catalog operation grounding*: selecting the right SDK client, method, and parameters across thousands of operations.
- **Architecture.** We propose *Operation-Visible Agent Architecture*, which combines centralized ReAct reasoning, parallel tool calling, and deterministic operation search, schema inspection, and validated invocation.
- **Evaluation.** We evaluate six configurations formed by crossing three agent-control patterns with two MCP tool-boundary designs, using multi-turn simulation, calibrated LLM-as-judge labels, and operational metrics.
- **Finding.** MCP tool-boundary design is a first-order factor: visible operation resolution improves task completion most in centralized ReAct settings, while parallel tool calling mainly improves efficiency, reducing tokens by 54.1% and median latency by 43.3% over the production-style baseline.

2 Related Work

Agent architectures: reasoning, acting, and decomposition. Recent LM-agent work improves multi-step behavior through explicit reasoning (Wei et al., 2022), interleaved reasoning with observations / actions (Yao et al., 2023), and separated planning and execution (ReWOO; Xu et al., 2023). Other systems delegate computation to programs (PAL; Gao et al., 2023), incorporate verbal feedback loops (Reflexion; Shinn et al., 2023), frame agents as augmented LMs combining parametric generation with tools, memory, and external modules (Mialon et al., 2023; Hu et al., 2026), or decompose agents into specialized components (Han et al., 2026). We study where operation-selection decisions reside when enterprise cloud agents operate over thousands of SDK operations.

Tool use, API selection, and executable retrieval. Prior work explores API retrieval and invocation through Toolformer, API-Bank, ToolBench, StableToolBench, and Gorilla (Schick et al., 2023; Li et al., 2023; Qin et al., 2024; Guo et al., 2024; Patil et al., 2024). Enterprise cloud agents operate over thousands of SDK operations, making operation selection a retrieval and grounding problem rather than simple function-calling (Lewis et al., 2020; Liu et al., 2026). Unlike document retrieval, operation retrieval must return executable contracts with valid parameters and execution constraints. We study how MCP tool boundaries expose operation-selection decisions in enterprise SDK catalogs.

MCP and production tool boundaries. The MCP standardizes tool access for LMs through a client-server interface (Karimova and Dadashova, 2025). Yet MCP does not determine where domain-specific intelligence should reside: in the prompt, a subagent, the tool server, or a deterministic middleware layer. In production cloud environments, this boundary must handle evolving schemas, multi-tenancy, token limits, and latency. We therefore study MCP tool-boundary design using task-completion and operational metrics, including tool errors, LLM calls, token usage, and latency.

3 Problem Setting

Let x_1 denote the initial user request and $T = \{t_i\}_{i=1}^{|T|}$ be a large enterprise tool catalog. Each operation $t_i \in T$ corresponds to an executable SDK client method and can be represented as $t_i = (c_i, m_i, d_i, P_i^{req}, P_i^{opt}, A_i, E_i)$, where c_i is the service client, m_i is the method name, d_i is a natural-language description, P_i^{req} and P_i^{opt} are required and optional parameters, A_i captures authorization constraints, and E_i contains execution metadata.

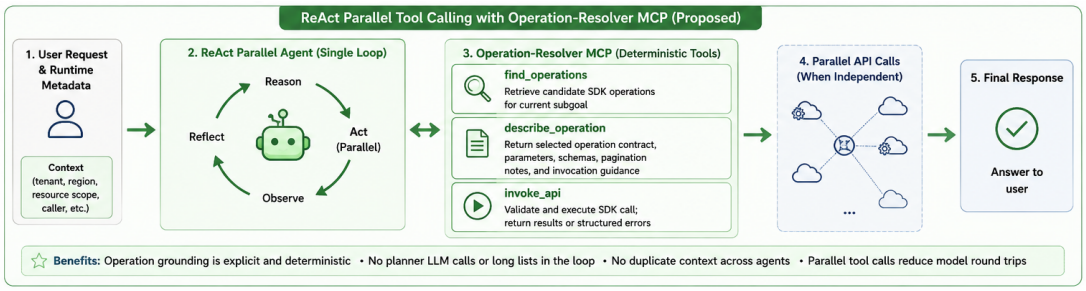

Figure 2: The Operation-Visible Agent Architecture: a ReAct Agent able to emit parallel tool calls to the Operation-Resolver MCP.

The agent operates in a multi-turn setting with dialogue history $H_j = (x_{1:j}, a_{1:j-1}, r_{1:j-1})$, where x_j is a user utterance or clarification, $a_j = (t_j, \theta_j)$ is a selected SDK operation t_j with grounded arguments θ_j, and r_j is the tool outputs. To satisfy the user's request, the agent must choose a sequence of operations $a_{1:k}$, construct valid inputs from user values, runtime metadata, and prior observations, execute the required calls, and produce a final answer y grounded in observed results.

The system optimizes for both task quality and efficiency. A successful run selects the correct SDK operation, constructs grounded parameters, avoids irrelevant schemas or broad operation lists, reduces LLM calls, tokens, and latency, recovers from recoverable API failures, and asks for clarification or approval when needed. We use three task-level labels: successful, partially successful, and failed. A run is fully successful if the user's goal is completed, partially successful if progress is made but blocked by missing information, unavailable resources, or permissions, and failed if a feasible path existed but the task was not completed.

4 System Design

We compare agent architectures along two dimensions: the agent-control pattern and the MCP tool boundary. For control, we consider lead-worker, sequential ReAct, and ReAct Parallel Tool Calling. Lead-worker decomposes the request into worker subtasks and later synthesizes their outputs; sequential ReAct keeps reasoning centralized but invokes one tool per model turn; and ReAct Parallel Tool Calling preserves the centralized ReAct trajectory while allowing one model turn to issue multiple independent tool calls when safe to execute in parallel.

Lead-worker with Tool-Planner-Driven MCP. The Tool-Planner-Driven MCP was introduced to reduce blind API guessing from model parametric knowledge. It exposes an LLM-based `planner` (Liu et al., 2025), `list_ops` for retrieving operations of a selected SDK client, and `invoke_api` for generic SDK execution. In the lead-worker workflow, the lead decomposes the task, the planner selects relevant clients and operations from compact client summaries and long operation lists, workers execute the selected calls, and the lead merges their results. This improves over unguided invocation, but remains costly: the planner needs additional LLM calls to ground each step to concrete operations, and lead-worker duplicates substantial context across the lead, planner, and workers. Thus, operation selection becomes more guided, but repeated listing calls, duplicated context, and planner-internal decisions can still increase token usage and context bloat. This design can remain competitive for broad inventory or troubleshooting tasks, but its operation grounding is more expensive and less visible.

Operation-Visible Agent Architecture. The Operation-Resolver MCP changes the MCP boundary by replacing internal LLM planning with operation-resolution tools: `find_operations` retrieves candidate SDK operations for the current subgoal; `describe_operation` returns the selected operation contract, parameters, schemas, pagination notes, and invocation guidance; and `invoke_api` validates and executes the grounded SDK call, returning results or structured errors. The resolver builds search results and operation descriptions from the installed SDK catalog without running an internal LLM planning dialogue. The main LLM still chooses candidate operations, fills arguments, and interprets results, keeping SDK knowledge in the MCP layer while making operation choice visible to the main ReAct loop.

Operation-Visible ReAct Parallel Workflow. The *Operation-Visible Agent Architecture* workflow keeps the reasoning trajectory in one ReAct loop: the agent receives the user request and runtime metadata, uses `find_operations` to retrieve candidate operations, inspects selected operations with `describe_operation`, constructs grounded arguments from user input, metadata, and prior observations, and executes validated calls through `invoke_api`. When multiple read or lookup calls have known arguments and no dependency, ReAct Parallel Tool Calling issues them in parallel in one model turn. The agent synthesizes results, recovers from individual API errors when possible, and asks for clarification or approval when required. The key distinction is not only parallel execution: the lead-worker planner hides operation grounding behind planner-internal LLM calls and duplicated context, whereas *Operation-Visible Agent Architecture* keeps operation candidates, schemas, validation feedback, and invocation results visible in the main ReAct trajectory.

4.1 Operation Retrieval and Resolution

The Operation-Resolver is deterministic in the specific sense that catalog retrieval, schema lookup, and argument validation require no internal LLM planner. It does not deterministically map a user utterance to a final correct operation. At turn j, the central agent constructs a search query q_j from the current subgoal and history H_j. `find_operations` retrieves a ranked candidate set

$$C_j = \mathrm{TopK}_{t_i \in T}\, s(q_j, z_i),$$

where z_i is the indexed representation of operation t_i. The central ReAct agent then selects candidates from C_j, inspects their contracts using `describe_operation`, grounds arguments, and invokes validated calls using `invoke_api`.

At turn j, operation resolution produces a candidate set $C_j \subseteq T$ from the current request and history. For a candidate $t_i \in C_j$, `describe_operation` exposes $(c_i, m_i, d_i, P_i^{req}, P_i^{opt}, A_i, E_i)$. The agent constructs arguments θ_j, after which `invoke_api`(t_i, θ_j) validates the arguments and returns an execution result r_j.

Internally, `find_operations` indexes each operation using its summary, client and service identifiers, operation and action terms, parameter signatures, API path, and invocation constraints. The local backend embeds documents and queries with `cohere.embed-v4.0`, and ranks candidates using $s(q, z_i) = \mathrm{cosine}(e_q, e_i) + 0.12\ell(q, z_i)$, where ℓ combines token overlap, operation coverage, action matching, and exact operation and client matches.

Across the analyzed search calls, 92.5% of searches followed by a downstream target contained at least one downstream operation in the top 20 candidates. Search reformulation occurs in 46.1% of targeted runs, and later searches recover 22.8% of targets missed by the first search. These results motivate keeping candidate retrieval and reformulation in the central reasoning trajectory; full retrieval statistics appear in Appendix E.3.

5 Evaluation

5.1 Dataset

We construct an expert-curated benchmark of 175 cloud-console tasks drawn from three complementary sources: product-management use cases, Regression use cases, and domain specific security/networking workflows. The benchmark evaluates whether agents can complete realistic operational tasks expressed as natural language, requiring interactions with a large catalog of cloud APIs. The benchmark contains proprietary enterprise cloud-console workflows derived from internal product-management artifacts and customer-reported issue reports, so the full dataset cannot be released. To support transparency, we provide anonymized task-card examples in Appendix A.

Each task is encoded as a task card consisting of an initial user query, a complexity label, cloud context, user goal, and success criteria. The tasks cover common cloud-console topics, including resource inventory, configuration lookup, cost and metrics, troubleshooting, identity and access management, security analysis. Complexity is categorized as *Easy*, *Medium*, or *Hard*. *Easy* tasks are retrieval-oriented tasks with relatively direct mappings between user intent and cloud operations. *Medium* tasks require contextual reasoning or interpretation across multiple resources. *Hard* tasks are advanced operational workflows requiring multi-step planning or transformational actions involving multiple dependent operations. The benchmark contains 55 *Easy* (31.4%), 76 *Medium* (43.4%), and 44 *Hard* (25.1%). Detailed distributions by source and complexity level are provided in Appendix B.

The benchmark includes 91 representative cloud-console workflows curated by product managers,

66 regression tasks reconstructed from 70 user-reported Jira issues, and 18 domain-specific security and networking tasks. The regression tasks capture real-world failures, edge cases, and complex system states, while the domain-specific tasks require more specialized operation selection. We run the full six-configuration comparison on the 91-task cloud-console benchmark. For the regression and domain-specific suites, we compare the production-style lead-worker Tool-Planner baseline against the proposed architecture.

5.2 Simulation Protocol and LLM-as-a-Judge

All runs use the same conversation harness. Each task begins with the task's `initial_query`; subsequent user turns are generated by an LLM-backed simulator conditioned on the task persona, private knowledge, chat history, and goal. A conversation stops when the simulator marks the goal as satisfied, the task-completion judge stops the run, or the turn budget is exhausted. The simulator and judge are held fixed across all configurations. The simulator is conditioned on task metadata and conversation history, but the evaluated agent never receives the success criteria or judge rationale.

We then evaluate each multi-turn trace with an LLM-as-a-judge (LLMaaJ). The judge receives the full trace, including conversation history, agent responses, tool calls, arguments, results, and tool or API errors, and assigns one of three labels: fully successful, partially successful, or failed. We tuned the judge rubric using trace inspections and calibration examples; Appendix D describes this process. The final judge matched human labels on 79.6% of examples. For the `fully_successful` class, it achieves 0.89 precision, 0.87 recall, and 0.88 F1 over 45 human-labeled examples (Table 7). Precision and recall are therefore reasonably balanced, reducing the concern that judge-labeled full success is driven primarily by systematic over-prediction. We nevertheless avoid over-interpreting small success-count differences and focus on larger task-level gaps. We report full-success outcomes as *judge-labeled* and interpret them alongside directly measured efficiency and reliability metrics, including LLM calls, token usage, latency, and tool-error rate.

5.3 Experimental Configurations

We evaluate six configurations formed by crossing three agent architectures (Lead-Worker, ReAct Sequential, ReAct Parallel) with two MCP tool-boundary designs (Tool-Planner-Driven and Operation-Resolver).

All configurations use the same model family, evaluation harness, user simulator, runtime environment, and judge. Planner-internal LLM calls are included in the reported LLM-call and token counts when they occur inside the evaluated architecture. This design isolates the effects of agent architecture and MCP tool-boundary design.

We evaluate performance in a fixed production-scale catalog containing approximately 8K operations. We do not vary catalog size and therefore do not estimate a scaling curve as operations or semantically similar distractors are added.

5.4 Metrics

We report each task as fully successful, partially successful, and failed. Efficiency metrics include LLM calls, input/output/total tokens, tool calls, and tool calls per LLM call. Reliability metrics include tool errors, sessions with tool errors, and tool-error rate. We distinguish tool errors from task failure because cloud APIs often return recoverable errors such as missing permissions, absent resources, unsupported parameters, or empty regional state.

6 Results

6.1 Cloud-Console Benchmark Results

Table 1 reports the six-configuration comparison on the 91-task Cloud-console benchmark. The strongest observed quality–efficiency trade-off is *Operation-Visible Agent Architecture*, which combines ReAct Parallel Tool Calling with Operation-Resolver and achieves 70 fully successful sessions, zero failed sessions, a 77% success rate, and the lowest median latency. The comparison shows a strong interaction between agent architecture and MCP tool-boundary design: Operation-Resolver provides the largest benefit when paired with a centralized ReAct loop.

The clearest comparison is within ReAct Parallel Tool Calling: replacing Tool-Planner-Driven with Operation-Resolver increases fully successful sessions from 53 to 70 while maintaining zero failed sessions. ReAct Sequential shows the same direction, improving from 58 to 62 fully successful sessions and reducing failures from five to zero. Thus, parallel execution alone is insufficient; it is most effective when paired with an MCP boundary that keeps operation selection explicit and visible.

Dataset	Agent Architecture	MCP Tool-Boundary Designs	Full Success	Partial Success	Fail	Success rate	LLM Calls	Total Tokens	Tool calls	Tool Err.	Med. lat (sec)
Cloud-console benchmark	Lead-worker	Tool-Planner-Driven	62	29	0	68%	1,795	116.6M	2,945	6.3%	298.0
	Lead-worker	Operation-Resolver	57	32	2	63%	1,631	219.0M	2,509	2.3%	177.2
	ReAct Sequential	Tool-Planner-Driven	58	28	5	64%	5,515	103.5M	**1,850**	2.2%	494.7
	ReAct Sequential	Operation-Resolver	62	29	0	68%	2,895	137.0M	2,746	**1.7%**	270.3
	ReAct Parallel Tool Calling	Tool-Planner-Driven	53	38	0	58%	4,389	**40.9M**	3,556	1.8%	360.1
	ReAct Parallel Tool Calling	Operation-Resolver	**70**	21	0	**77%**	**1,117**	53.5M	4,200	2.1%	**168.9**

Table 1: Main evaluation on the 91-task Cloud-console benchmark across agent-control architectures and MCP tool-boundary designs. The table reports both task-quality outcomes and efficiency metrics. The best overall configuration is *Operation-Visible Agent Architecture*, which achieves the highest full-success rate and lowest median latency while using substantially fewer LLM calls and tokens than sequential ReAct on the same Operation-Resolver boundary. "Med. lat (sec)" denotes median latency in seconds. Task outcomes are assigned by the calibrated LLM judge; operational metrics are measured directly from execution traces.

Dataset	Agent Architecture	MCP Tool-Boundary Designs	Succ.	Part. Succ.	Fail	Succ. rate
Regression use cases	Lead-worker	Tool-Planner-Driven	32	28	6	48%
	ReAct Parallel Tool Calling	Operation-Resolver	**36**	29	**1**	**55%**
Domain-specific	Lead-worker	Tool-Planner-Driven	9	8	1	50%
	ReAct Parallel Tool Calling	Operation-Resolver	**12**	6	**0**	**67%**

Table 2: Generalization on two additional suites: Jira-derived regression and domain-specific tasks. Compared with the production-style lead-worker Tool-Planner baseline, ReAct Parallel Tool Calling with Operation-Resolver improves full-success rate and reduces failed sessions in both.

6.2 Parallel Tool-Calling Efficiency Gain

Table 1 isolates the efficiency gains from parallel execution under the same Operation-Resolver MCP boundary. Compared with ReAct Sequential + Operation-Resolver, *Operation-Visible Agent Architecture* increases successful sessions from 62 to 70 while reducing LLM calls by 61.4% (2,895 to 1,117), total tokens by 61.0% (137.0M to 53.5M), and median latency by 37.5% (270.3s to 168.9s).

Tool calls also increase from 2,746 to 4,200 (+52.9%), reflecting fewer model turns issuing larger tool-invocation batches. We analyze these additional tool calls and operation-retrieval behavior in Appendix E.3. The Tool-Planner-Driven parallel configuration does not achieve comparable task-success gains, indicating that parallelism improves efficiency most reliably when operation grounding is visible and well-structured.

6.3 Regression Use Cases

We use the additional suites as targeted regression checks rather than full factorial architecture sweeps. We evaluate the Agent Configurations on the Regression Use Cases. Compared with the Cloud-console benchmark, these tasks are more challenging because they originate from real user-reported failures and edge cases. Consequently, success rates are lower across all Agent Configurations than on the Cloud-console benchmark.

Despite the increased difficulty, the *Operation-Visible Agent Architecture* achieves the strongest overall performance (Table 2), completing 36 of 66 (55%) tasks successfully while reducing failed sessions from six to one relative to the *Lead-Worker* baseline. The gain is modest but directionally consistent, with full success increasing from 32 to 36 and failed sessions decreasing from six to one. These results provide directional evidence that the same pattern holds beyond representative workflows and remain valuable under regression-oriented scenarios.

6.4 Domain-Specific Security and Networking

The Domain-Specific dataset evaluates whether the benefits transfer to specialized domains, e.g., security and networking workflows. Unlike the Cloud-console benchmark, these tasks require deeper domain knowledge and more precise selection among semantically similar operations, making accurate operation grounding particularly important.

The *Operation-Visible Agent Architecture* successfully completes 12 of 18 tasks (67%), compared with 9 of 18 (50%) for the Lead-Worker baseline. Although the dataset is small, the results suggest that the benefits of operation visibility extend beyond representative cloud-management workflows to specialized operational domains.

Taken together, the Cloud-console benchmark, Regression Use Cases, and Domain-Specific dataset evaluate representative workflows, failure-

driven scenarios, and specialized operational expertise. The *Operation-Visible Agent Architecture* achieves the strongest performance across all three settings, suggesting that operation visibility remains useful in specialized domains, though the small sample size means this result should be treated as supportive rather than conclusive.

7 Lessons for Enterprise Agent Design

The tested MCP boundary materially affects the quality-efficiency trade-off. The Operation-Resolver boundary is most effective when paired with centralized ReAct control. This boundary jointly changes focused retrieval, schema exposure, validation feedback, internal planner calls, and the visibility of operation-selection evidence.

Operation-visible resolution provides an inspectable interface. Exposing candidates, schemas, validation feedback, and invocation results allows the central agent to reformulate searches and recover from errors. The experiments support the Operation-Resolver boundary as a whole; they do not isolate visibility from retrieval, schema exposure, validation, or the removal of planner-internal LLM calls.

Parallel tool calling is primarily an efficiency mechanism. Parallel execution reduces model round trips and latency by allowing independent API calls to be executed concurrently. However, parallelism alone does not guarantee higher task success; its benefits are greatest when combined with effective operation grounding.

8 Conclusion

Enterprise agents operating over large API ecosystems face a challenge beyond traditional tool-use settings: grounding user requests to correct operations across large catalogs of services, APIs, schemas, and execution constraints. Across a 91-task six-configuration benchmark and two targeted evaluation suites, we find that MCP tool-boundary design is a first-order design choice for such agents. The largest gains occur when operation discovery, schema inspection, and invocation remain visible within the agent's reasoning process. Parallel tool calling primarily improves efficiency, while operation visibility improves task success. These results suggest operation grounding should be treated as a systems concern. As enterprise tool ecosystems grow, effective architectures must balance abstraction with transparency, enabling agents to reason about operation choice, recover from failures, and coordinate complex workflows using structured retrieval and validation mechanisms.

Limitations

The main limitation is reproducibility. While benchmark specifications and anonymized task-card examples are shared, the cloud environment, proprietary simulator, and runtime state cannot be released which means external researchers cannot reproduce the evaluation environment.

The evaluation relies on an LLM-backed user simulator and an LLMaaJ pipeline. Although the judge was calibrated through manual review and achieved 79.6% agreement with human annotations (Appendix D), simulated users and automated evaluation remain imperfect proxies for real enterprise users and human judgement. Judge-based metrics should therefore be interpreted as approximate indicators of task completion, particularly for small performance differences.

The evaluation demonstrates operation grounding in a thousands-operation setting but does not establish how performance changes with catalog size. Controlled catalog-size and distractor-operation experiments remain future work.

The Operation-Resolver boundary jointly changes retrieval, schema exposure, validation, internal planner calls, and evidence visibility. Because we do not hold these components fixed in a visible-versus-hidden ablation, the observed gains cannot be attributed to visibility alone.

All experiments use a single model family (GPT-5.4) within one enterprise cloud ecosystem. We therefore interpret the results as architectural rather than foundation-model comparisons. While the observed trends were consistent across configurations, the magnitude of improvements may differ across model families, cloud providers or environments.

Ethical Considerations

The cloud agent evaluated in this work interacts with sensitive enterprise metadata, resource identifiers, and user logs. To protect data privacy, we do not release any raw datasets, traces, or user data. All evaluation metrics and examples presented in this paper are strictly aggregated or abstracted to protect proprietary information. Any descriptive examples have been fully anonymized to remove company names, product names, internal project paths, and customer-specific resource identifiers.

References

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: Program-aided language models. In *International Conference on Machine Learning*, volume 202, pages 10764–10799.

Sandip Ghoshal, Anshul Mittal, Jyotika Singh, Miguel Ballesteros, Weiyi Sun, Fang Tu, Shailender Singh, Yassine Benajiba, Fahad Shah, Sujeeth Bharadwaj, Sujith Ravi, and Dan Roth. 2026. JTPRO: A joint tool–prompt reflective optimization framework for language agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 40573–40595, San Diego, California, United States. Association for Computational Linguistics.

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. 2024. StableToolBench: Towards stable large-scale benchmarking on tool learning of large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 11143–11156, Bangkok, Thailand. Association for Computational Linguistics.

Kevin Han, Siddharth Maddikayala, Tim Knappe, Om Patel, Austen Liao, and Amir Barati Farimani. 2026. TDFlow: Agentic workflows for test driven development. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1511–1527, Rabat, Morocco. Association for Computational Linguistics.

Jinchao Hu, Meizhi Zhong, Kehai Chen, Xuefeng Bai, and Min Zhang. 2026. Agentic tool use in large language models. *Preprint*, arXiv:2604.00835.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*.

Sevinj Karimova and Ulviya Dadashova. 2025. The model context protocol: A standardization analysis for application integration. *UNEC Journal of Computer Science and Digital Technologies*, 1(1):50–59.

Sameer Komoravolu and Khalil Mrini. 2026. Agent-testing agent: A meta-agent for automated testing and evaluation of conversational AI agents. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7199–7214, Rabat, Morocco. Association for Computational Linguistics.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Kuttler, Mike Lewis, Wen-tau Yih, Tim Rocktaschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474.

Junjie Li, Xi Xiao, Yunbei Zhang, Chen Liu, Lin Zhao, Xiaoying Liao, Yingrui Ji, Janet Wang, Jianyang Gu, Yingqiang Ge, Weijie Xu, Xi Fang, Xiang Xu, Tianchen Zhao, Youngeun Kim, Tianyang Wang, Jihun Hamm, Smita Krishnaswamy, Jun Huan, and Chandan Reddy. 2026. Agent harness engineering: A survey.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. API-bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3102–3116, Singapore. Association for Computational Linguistics.

Hanbing Liu, Chunhao Tian, Nan An, Ziyuan Wang, Pinyan Lu, Changyuan Yu, and Qi Qi. 2026. Budget-constrained agentic large language models: Intention-based planning for costly tool use. *Preprint*, arXiv:2602.11541.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, and 3 others. 2024. Agentbench: Evaluating LLMs as agents. In *The Twelfth International Conference on Learning Representations*.

Yanming Liu, Xinyue Peng, Jiannan Cao, Shi Bo, Yuwei Zhang, Xuhong Zhang, Sheng Cheng, Xun Wang, Jianwei Yin, and Tianyu Du. 2025. Tool-planner: Task planning with clusters across multiple tools. In *The Thirteenth International Conference on Learning Representations*.

Gregoire Mialon, Roberto Dessi, Maria Lomeli, Christoforos Nalmpantis, Ramakanth Pasunuru, Roberta Raileanu, Baptiste Roziere, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. 2023. Augmented language models: A survey. *Transactions on Machine Learning Research*.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2024. Gorilla: large language model connected with massive apis. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA. Curran Associates Inc.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *International Conference on Learning Representations*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023.

Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. 2024. MINT: Evaluating LLMs in multi-turn interaction with tools and language feedback. In *International Conference on Learning Representations*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35.

Binfeng Xu, Zhiyuan Peng, Bowen Lei, Subhabrata Mukherjee, Yuchen Liu, and Dongkuan Xu. 2023. ReWOO: Decoupling reasoning from observations for efficient augmented language models. *Preprint*, arXiv:2305.18323.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik R Narasimhan. 2025. $\{\tau\}$-bench: A benchmark for tool-agent-user interaction in real-world domains. In *The Thirteenth International Conference on Learning Representations*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. WebArena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*.

A Dataset Specification

Task Card – Anonymized Each benchmark instance is represented as a task card that specifies the user goal, anonymized cloud context, and success criteria. The task card serves as the input specification for the multi-turn evaluation: the simulator uses it to generate realistic user behavior and to determine when the conversation should stop. Table 3 shows an illustrative anonymized example.

B Dataset Composition and Complexity Distribution

Table 4 summarizes the benchmark composition by source and complexity level. The dataset combines three complementary sources: 91 cloud-console benchmark tasks covering representative enterprise cloud workflows, 66 regression use cases derived from failure-driven and edge-case scenarios, and 18 domain-specific tasks focused on specialized security and networking workflows. The cloud-console benchmark contains a balanced mix of medium and hard tasks, while the regression set is concentrated in easy and medium cases. The domain-specific subset is smaller but captures expert-level scenarios that require more specialized cloud knowledge.

C SDK Client and Service Usage Statistics

We analyze executable SDK API calls issued by the proposed *Operation-Visible Agent Architecture* configuration on the 91-task Cloud-console benchmark. We count calls to the anonymized `invoke_api` tool, which corresponds to the concrete SDK API execution tool. We exclude operation-grounding calls such as `find_operations` and `describe_operation`, because those measure operation retrieval and schema inspection rather than executable cloud API use. A call is counted as successful when the returned result contains neither a parsed API error nor a tool-level error.

Table 5 summarizes the executable API usage profile. Across 91 tasks, *Operation-Visible Agent Architecture* issued 3,095 `invoke_api` calls, covering 96 distinct client-operation pairs across 22 SDK clients. The median task issued 18 executable API calls when zero-call tasks are included, and 19 among the 87 tasks that issued at least one executable call. This indicates that many benchmark tasks require multi-step API interaction rather than a single isolated invocation.

Field	Anonymized example
Task domain	Cloud Resource Management
User goal	Identify stopped compute instances in a development compartment and restart the testing instance.
Initial user request	"Can you help me find the stopped instances in my development compartment and start the one used for testing?"
Anonymized context	The tenancy contains multiple compartments. The target compartment is denoted as <COMPARTMENT_A>. The target instance is denoted as <INSTANCE_TEST>.
Complexity	Medium
Success criteria	The agent identifies the correct compartment and stopped testing instance, starts the correct instance , and provides a concise confirmation to the user.

Table 3: An anonymized task card example.

Source	Description	# Easy	# Medium	# Hard	Total
Cloud-console benchmark	Representative enterprise cloud workflows.	23	34	34	91
Regression use cases	Failure-driven and edge-case scenarios derived from real user issues.	24	37	5	66
Domain-specific	Specialized security and networking workflows requiring domain expertise.	9	5	4	18

Table 4: Dataset composition by source and complexity label.

Metric	Value
Task records analyzed	91
Total `invoke_api` calls	3,095
Successful `invoke_api` calls	3,006
Distinct client-operation pairs	96
Distinct SDK clients	22
Median `invoke_api` calls / task	18

Table 5: Executable SDK API usage statistics for *Operation-Visible Agent Architecture* on the 91-task Cloud-console benchmark. The table counts only anonymized `invoke_api` calls and excludes operation-search and operation-description calls.

Table 6 breaks down executable API usage by task complexity. We report medians because call counts are right-skewed: a small number of broad inventory or environment-limited tasks require many additional calls to locate relevant resources or confirm that requested resources are absent. In this run, medium tasks have the highest median call volume, reflecting broad discovery and inventory workloads in that slice of the benchmark.

Complexity	Tasks	Total calls	Median calls/task
Easy	23	541	3
Medium	34	1,742	25.5
Hard	34	812	14.5

Table 6: Executable `invoke_api` calls by task complexity for *Operation-Visible Agent Architecture*. Medians include tasks with zero executable API calls.

The observed executable API usage spans multiple service families, including compute, networking, storage, resource search, identity and access management, object storage, database, migration, Fusion applications, monitoring, and cost analysis. This supports the benchmark's role as a large-toolset evaluation: the agent must ground user requests to concrete SDK operations across a broad API surface, rather than repeatedly solving variants of a single service-specific task.

D Evaluation Method Calibration

Agent evaluation and real-world environments. Evaluating multi-step agents requires interactive environments and functional completion metrics rather than static text similarity (Liu et al., 2024; Zhou et al., 2024). MINT highlights that isolated capabilities rarely predict multi-turn performance (Wang et al., 2024), SWE-bench emphasizes realistic tasks from production codebases and fully executable runtimes (Jimenez et al., 2024), and Agent-Testing Agent automates adaptive, persona-driven adversarial evaluation through a meta-agent (Komoravolu and Mrini, 2026). Our evaluation protocol extends this philosophy to live cloud APIs, analyzing task success alongside real-world ambient failures, missing contextual metadata, and resource constraints. To parse complex execution traces, we leverage an LLMaaJ paradigm, validating its reliability against human-annotated baseline alignments detailed below.

We used an LLM-based judge to evaluate task

completion quality from agent interaction traces. For reporting in the main paper, we collapse the judge outputs into three categories: fully successful, failed, and partially successful. The partially successful category captures all outcomes that do not cleanly fall into either complete success or outright failure, including intermediate or ambiguous cases.

Judge prompt development and calibration We developed the judge prompt through an iterative calibration process. The initial prompt was designed to evaluate agent traces conservatively and evidence-based, producing structured judgments at both the turn level and the overall session level. The judge was instructed to identify whether a user request was actionable and whether the agent fully satisfied the request, failed to satisfy it, or produced an outcome that was only partially successful or ambiguous.

The prompt was first tuned using manual assessment on a small held-out calibration subset of approximately 50 spot-checked samples. These samples were single-turn interactions and were not part of any datasets used for the final reported evaluation. During calibration, we inspected examples where the judge disagreed with human assessment and refined the rubric to improve consistency with human judgments.

After this initial calibration, we applied the judge to a separate multi-turn dataset (curated by product managers with domain expertise based on observed user behavior) and performed an additional manual review of individual examples (by 5 individuals with science and engineering backgrounds with up to 3 labels collected per sample) to compare judge performance with human assessment. Only unanimous fully_successful cases were deemed fully successful, and majority vote was taken for other labels. For samples with conflicting labels and no clear majority, another labeler reviewed and labeled the sample to resolve the conflict.

Human alignment study We measured agreement between the final LLM judge and human labels on the held-out multi-turn validation set. The judge achieved 79.6% overall agreement with the human labels. In particular, the `fully_successful` class has 0.89 precision, 0.87 recall, and 0.88 F1 across 45 human-labeled examples (Table 7). Performance is also reasonably balanced for `partially_successful` and `failed`, with F1 scores of 0.76 and 0.73, respectively.

Label aggregation for reported results For the headline evaluation numbers in the paper, we map the judge outputs into three categories: fully_successful, failed, and partial_success. The partial_success category includes cases where the task was only partially completed, the outcome was ambiguous, or the interaction did not cleanly correspond to either full success or failure.

This aggregation serves two purposes. First, it aligns the reported metrics with the primary evaluation question: whether the system fully completed the user request, failed to complete it, or landed somewhere in between. Second, it avoids over-interpreting fine-grained categories whose boundaries may be less stable across examples.

Interpretation and limitations These results suggest that the LLM judge provides a reasonable but imperfect proxy for human evaluation. The alignment study shows balanced precision and recall for the `fully_successful` class, reducing the risk that judge-labeled full success is driven primarily by over-prediction. However, the judge remains an automated proxy rather than a replacement for human evaluation. We therefore interpret judge-labeled task-quality differences together with paired statistical tests and directly measured operational metrics. The calibration samples used to tune the judge prompt were distinct from the datasets used for final evaluation, reducing the risk of overfitting the judge to the reported evaluation set. At the same time, the observed human-judge agreement motivates caution when interpreting small differences between systems or configurations.

E Appendix: Additional Analysis

E.1 Statistical Analysis of Agent Configurations

We perform paired statistical tests over the same 91 Cloud-console benchmark tasks. For task-quality outcomes, we use an exact two-sided McNemar test on the binary event `fully_successful` versus not fully successful. For efficiency, we use a two-sided Wilcoxon signed-rank test on paired per-task token totals. We report Holm-adjusted p-values across the 15 pairwise comparisons. For Lead-worker + Tool-Planner-Driven MCP, planner-internal token telemetry is included in the aggregate token total, but it is only available at run level; the Wilcoxon test therefore uses trace-visible per-task token counts for that configuration. The results are in Table 9 and the counts in Table 8.

Class	Precision	Recall	F1	Support
partially_successful	0.77	0.75	0.76	36
failed	0.71	0.75	0.73	32
fully_successful	0.89	0.87	0.88	45

Table 7: Per-class performance between the LLM judge and human labels on the matched multi-turn validation set.

Code	Configuration	Succ.	Part.	Fail	Tokens
LW-OR	Lead-worker + Operation-Resolver MCP	57	32	2	219.0M
RS-OR	ReAct Sequential + Operation-Resolver MCP	62	29	0	137.0M
RP-OR	ReAct Parallel Tool Calling + Operation-Resolver MCP	70	21	0	53.5M
LW-TP	Lead-worker + Tool-Planner-Driven MCP	62	29	0	116.6M
RS-TP	ReAct Sequential + Tool-Planner-Driven MCP	58	28	5	103.5M
RP-TP	ReAct Parallel Tool Calling + Tool-Planner-Driven MCP	53	38	0	40.9M

Table 8: Outcome counts and total token usage for the six 91-task configurations. Part. aggregates non-failed partial-success outcomes.

The token-usage tests show that parallel tool calling is consistently more efficient. Under the same Operation-Resolver MCP boundary, ReAct Parallel Tool Calling uses significantly fewer tokens than ReAct Sequential after Holm correction ($p = 4.60 \times 10^{-6}$), while also increasing fully successful sessions from 62 to 70. The McNemar test for this quality difference is directionally favorable but not significant after correction ($p_{\text{Holm}} = 1.0000$), reflecting the limited sample size and the judge-calibration caveat discussed in Appendix D.

The only full-success comparison that remains significant after Holm correction is ReAct Parallel Tool Calling + Operation-Resolver MCP versus ReAct Parallel Tool Calling + Tool-Planner-Driven MCP: 23 tasks are fully successful only with the operation-visible boundary, compared with 6 only with the planner-driven boundary ($p_{\text{Holm}} = 0.0347$). This supports the main conclusion that parallel execution is primarily an efficiency mechanism, while visible operation resolution is the factor that turns that efficiency into higher task completion.

E.2 Reasoning-Effort Ablation

Table 10 reports a reasoning-effort ablation for the *Operation-Visible Agent Architecture*. Reducing reasoning effort decreases token usage substantially but also reduces task completion quality. Medium reasoning achieves 77% (70/91) fully successful sessions, compared with 62% (56/91) for low reasoning and 46% (42/91) for no reasoning.

The reduction in quality is accompanied by increases in partially successful outcomes and failed sessions, indicating that lower reasoning settings often struggle with operation selection, recovery, and multi-step coordination. While lower reasoning settings reduce inference cost, the resulting degradation in task completion suggests that reasoning remains an important component of enterprise cloud-agent performance, even when operation grounding is improved.

E.3 Operation Retrieval Analysis

We analyze `find_operations`, the operation-search tool used by the Operation-Resolver MCP. For each `find_operations` call, we parse the ranked candidate list and infer the downstream target operation from the next `describe_operation` or `invoke_api` call before the next search. This is not a human-authored answer key; it measures whether search surfaced the operation that the agent later chose.

Across 406 search calls from 91 sessions, the tool returned 8,120 candidate rows covering 1,608 unique operations. We identified 2,277 downstream target instances: 456 `describe_operation` targets and 1,821 `invoke_api` targets. Overall Recall@20 is 55.7% at the target-instance level, with MRR 0.345. Recall is higher for operations the agent chose to inspect with `describe_operation` than for operations it later invoked, suggesting that operation search is more reliable for narrowing candidate operations than for covering every downstream execution call.

At the search-call level, coverage is higher: among searches followed by downstream targets, 92.5% contained at least one downstream operation in the top 20 candidates. Search reformulation is common: 46.1% of targeted search runs required more than one `find_operations` call, and later searches recovered 22.8% of targets that were missed by the first search. Retrieval quality also varies by task difficulty: Recall@20 is 61.3% for easy tasks, 60.7% for medium tasks, and 48.2% for hard tasks.

These results support the design choice of keeping operation search visible to the main ReAct loop. `find_operations` often provides useful candidates, but it is not perfect; the agent benefits from being able to reformulate searches, inspect operation descriptions, and recover from retrieval misses

Pair	Lower-token run	W	Wilcoxon p	Holm p	Succ. A/B	A-only/B-only	McNemar p	Holm p
LW-OR vs RS-OR	RS-OR	1399.0	0.0060	0.0241	57/62	10/15	0.4244	1.0000
LW-OR vs RP-OR	RP-OR	225.0	1.43×10^{-13}	2.00×10^{-12}	57/70	7/20	0.0192	0.2682
LW-OR vs LW-TP	LW-TP	671.0	1.82×10^{-8}	2.19×10^{-7}	57/62	8/13	0.3833	1.0000
LW-OR vs RS-TP	RS-TP	1126.0	1.30×10^{-4}	0.0010	57/58	18/19	1.0000	1.0000
LW-OR vs RP-TP	RP-TP	116.0	5.09×10^{-15}	7.63×10^{-14}	57/53	22/18	0.6358	1.0000
RS-OR vs RP-OR	RP-OR	819.0	4.60×10^{-7}	4.60×10^{-6}	62/70	5/13	0.0963	1.0000
RS-OR vs LW-TP	LW-TP	1676.0	0.0989	0.1977	62/62	8/8	1.0000	1.0000
RS-OR vs RS-TP	RS-TP	1508.0	0.0206	0.0618	62/58	17/13	0.5847	1.0000
RS-OR vs RP-TP	RP-TP	626.0	6.39×10^{-9}	8.31×10^{-8}	62/53	19/10	0.1360	1.0000
RP-OR vs LW-TP	RP-OR	1139.0	1.59×10^{-4}	0.0011	70/62	14/6	0.1153	1.0000
RP-OR vs RS-TP	RP-OR	1260.0	9.77×10^{-4}	0.0059	70/58	19/7	0.0290	0.3765
RP-OR vs RP-TP	RP-TP	1355.0	0.0035	0.0174	70/53	23/6	0.0023	0.0347
LW-TP vs RS-TP	RS-TP	2088.0	0.9842	0.9842	62/58	18/14	0.5966	1.0000
LW-TP vs RP-TP	RP-TP	803.0	3.30×10^{-7}	3.63×10^{-6}	62/53	21/12	0.1628	1.0000
RS-TP vs RP-TP	RP-TP	984.0	1.14×10^{-5}	1.02×10^{-4}	58/53	15/10	0.4244	1.0000

Table 9: Pairwise Wilcoxon signed-rank tests for token usage and exact McNemar tests for fully successful outcomes. "A-only/B-only" gives discordant fully successful counts for the first and second configuration in each pair.

Reasoning	Succ.	Part. Succ.	Fail	Succ. rate	Tokens
Medium	70	21	0	77%	53.5M
Low	56	33	2	62%	34.9M
None	42	48	1	46%	26.3M

Table 10: Reasoning-effort ablation for *Operation-Visible Agent Architecture* on the Cloud-console benchmark.

Target type	N	R@1	R@3	R@5	R@10	R@20	MRR
Any target	2,277	26.3	40.9	44.9	51.6	55.7	0.345
Describe target	456	37.5	50.0	53.5	59.6	65.8	0.450
Invoke target	1,821	23.4	38.7	42.7	49.6	53.2	0.318

Table 11: Target-instance Recall@K for `find_operations`. Values are percentages except MRR.

before invoking APIs.

E.4 Efficiency Gains from Parallel Tool Calling

To isolate the effect of parallel execution, Table 12 compares ReAct Sequential and ReAct Parallel Tool Calling under the same Operation-Resolver MCP Tool-Boundary Design. Parallel execution increases fully successful sessions from 62 to 70 while reducing LLM calls by 61.4% (2,895 to 1,117), total tokens by 61.0% (137.0M to 53.5M), and median latency by 37.5% (270.3s to 168.9s).

At the same time, tool calls increase from 2,746 to 4,200 (+52.9%), showing that the system performs more concrete API work with fewer model interactions. Thus, parallel tool calling primarily improves efficiency by amortizing model round trips across independent API operations. However, parallelism alone is insufficient: Tool-Planner-Driven parallel configurations reduce some overhead but do not achieve comparable task-success gains, reinforcing MCP Tool-Boundary Design as the main driver of quality.

Metric	Seq.	Parallel	Change
Fully successful	62	70	+8
LLM calls	2,895	1,117	-61.4%
Total tokens	137.0M	53.5M	-61.0%
Tool calls	2,746	4,200	+52.9%
Avg. latency	392.2s	217.4s	-44.6%
Median latency	270.3s	168.9s	-37.5%

Table 12: Effect of parallel tool calling under the same Operation-Resolver MCP boundary. Parallel execution increases concrete API work while reducing model calls, tokens, and latency.

E.5 Production Baseline vs. Proposed Architecture

Table 13 compares the production-style Lead-worker + Tool-Planner-Driven baseline against the proposed *Operation-Visible Agent Architecture*. The proposed architecture increases fully successful sessions from 62 to 70, while reducing LLM calls by 37.8%, total tokens by 54.1%, tool errors by 52.4%, and median latency by 43.3%. Although the proposed system issues more tool calls, it performs more concrete API work with fewer model round trips and substantially lower token cost. This comparison highlights the practical production tradeoff: replacing hidden planner-mediated operation grounding with visible Operation-Resolver grounding improves both task completion and operational efficiency.

E.6 Tool errors are not task failures.

The *Operation-Visible Agent Architecture* has more raw tool calls and 89 tool errors on the Cloud-console benchmark, but only 21 sessions with

Metric	Lead-worker + Tool-Planner	ReAct Parallel + Opn Resolver	Change
Fully successful	62	70	+12.9%
Success rate	68%	77%	+8.8 pp
Partially successful	29	21	-27.6%
Failed	0	0	0
LLM calls	1,795	1,117	-37.8%
Total tokens	116.6M	53.5M	-54.1%
Tool calls	2,945	4,200	+42.6%
Tool errors	187	89	-52.4%
Tool-error rate	6.3%	2.1%	-4.2 pp
Median latency	298.0s	168.9s	-43.3%

Table 13: Production-style baseline versus the proposed architecture on the 91-task Cloud-console benchmark. The proposed *Operation-Visible Agent Architecture* improves full-success rate while substantially reducing LLM calls, total tokens, tool errors, and median latency. "pp" denotes percentage-point change.

tool errors and no failed sessions. The production lead-worker + Tool-Planner-Driven MCP run has a higher error rate and 47 sessions with tool errors. For cloud agents, an API error may be a useful observation: no permission, no resource, invalid parameter, or wrong region can guide recovery. Evaluations should therefore track both raw errors and whether they lead to task-level failure.

E.7 Error Categories

We qualitatively reviewed internal trace reports to identify recurring error patterns in large-toolset enterprise agent runs. The categories in Table 14 are not mutually exclusive: a single unsuccessful or partially successful run may involve multiple issues, such as broad discovery followed by invalid parameters or missing user information. The main purpose of this taxonomy is to connect observed failures to architecture choices. Several categories arise when operation selection is either hidden behind planner or specialist calls, or when broad operation lists are exposed directly to the agent. These errors motivate the Operation-Resolver MCP design, which keeps operation candidates, operation contracts, validation feedback, and invocation results visible in the main ReAct trajectory. The taxonomy also highlights why raw tool-error counts should not be treated as task failures: some API errors are recoverable observations that help the agent identify missing permissions, absent resources, wrong regions, or invalid arguments.

E.8 Lead-worker decomposition still has value.

The production lead-worker baseline remains competitive on the Regression use cases, with 32 fully successful sessions out of 66. This suggests that explicit decomposition can still help in some broad inventory and troubleshooting tasks. A practical future direction is to keep the operation-visible ReAct loop, but add lightweight self-checks for tasks that require broad coverage across compartments, regions, or resource surfaces.

E.9 Tool-Call Mix

Table 15 reports the main tool-call inventory for the Cloud-console benchmark. The resolver variants expose operation search and description directly. The Tool-Planner-Driven lead-worker baseline exposes planner and broad operation-listing tools.

F Codex Agent with CLI or Terraform Style Tool Invocation under Controlled Execution

We explored an alternative formulation in which a Codex agent maps tool-use decisions to command completion over CLI or Terraform style interfaces. Instead of requiring the model to directly select among SDK-style tools and synthesize structured API invocations, this formulation asks the agent to generate executable command specifications that expose equivalent underlying capabilities. For console-oriented workloads, such command-centric interfaces are plausible and may show promising results, as they reduce tool use to a familiar textual generation problem and can simplify argument selection relative to deeply structured tool schemas. However, in enterprise settings, evaluation cannot be limited to task accuracy alone. Security posture, operational reliability, and governance constraints become central, particularly when model outputs are routed through Bash or infrastructure-as-code execution paths. A production deployment would therefore require a clear separation between agentic planning and execution, together with tight controls such as command allowlisting, argument validation, least-privilege execution, dry-run validation, audit logging, and human approval for sensitive actions. Thus, while Codex-agent-based command completion may improve tool-use accuracy for certain workloads, its practical adoption depends on robust safeguards around execution.

Category	Description	Mitigation
Broad discovery without execution	Planner or specialist path searches broadly, lists large catalogs, or suggests follow-up queries instead of producing the requested artifact.	Keep operation choice visible in the main loop.
Invalid parameter construction	Agent guesses parameter names, passes unsupported kwargs, or uses incompatible fields.	Require operation descriptions and validation feedback before invocation.
Large operation-list context	Listing full client operations expands context and can truncate useful evidence.	Retrieve focused operation candidates and operation dossiers.
Ungrounded identifiers	Agent uses malformed, stale, or invented resource identifiers.	Track identifier provenance from metadata, user input, tool results, and validator feedback.
Recoverable API errors	Individual calls fail due to authorization, absence, region, or parameter constraints, but another path can still satisfy the task.	Track session-level recovery, not only raw error count.
Missing information	Required input is absent or cannot be inferred from grounded context.	Ask targeted clarification rather than guessing.
Low-value parallel calls	Parallel batches include calls that could have been filtered before dispatch.	Use operation descriptions and validation to prune impossible calls.

Table 14: Qualitative error categories observed in internal trace analyses.

System	Find	Describe	Planner	List ops	Invoke	Total	Errors	Sessions w/ errors
Lead-worker + Tool-Planner-Driven MCP	0	0	69	394	2,482	2,945	187	47
Lead-worker + Operation-Resolver MCP	496	559	0	0	1,454	2,509	57	24
ReAct Sequential + Operation-Resolver MCP	437	526	0	0	1,783	2,746	46	22
Operation-Visible Agent Architecture	525	580	0	0	3,095	4,200	89	21

Table 15: Tool-call mix and error counts for the Cloud-console benchmark.

G Runtime Controls and Sources of Variance

All configurations use the same model family, task cards, conversation harness, turn budget, simulator, judge, and runtime environment. Configurations were executed in fixed order. Cloud state was reset between configurations. API responses were live. Transient backend failures were handled by the underlying production runtime; the evaluation introduced no additional retry mechanism. Simulator turns were generated independently for each configuration. Consequently, some run-to-run variation may remain.

H Extended Related Work

We position our work at the intersection of language model (LM) agents, tool-use optimization, and architectural systems design. Unlike prior work evaluating reasoning or API calling in isolation, we study the architectural boundary needed when a deployed enterprise agent operates over cloud SDK catalogs. In this regime, operation selection becomes a high-dimensional retrieval, grounding, and infrastructure optimization problem, not only a semantic reasoning task. We organize the literature around agent decomposition, tool selection, retrieval over executable contracts, protocol-level boundaries, and real-world evaluation.

Reasoning, acting, and agent decomposition. Chain-of-thought prompting showed that intermediate reasoning traces improve multi-step inference (Wei et al., 2022), and ReAct extended this paradigm by interleaving thoughts with environmental observations (Yao et al., 2023). Subsequent augmented-LM patterns decouple reasoning from execution (ReWOO; Xu et al., 2023), delegate symbolic computation to programs (PAL; Gao et al., 2023), or incorporate verbal reinforcement (Reflexion; Shinn et al., 2023). Broad surveys frame these methods as modular extensions that fuse parametric generation with external tools and memory (Mialon et al., 2023; Hu et al., 2026), while related work decomposes agents into hierarchical planners, routers, or specialized subagents (Han et al., 2026). Such multi-agent patterns isolate complex subtasks but raise architectural questions about which granular decisions remain visible to the primary reasoning

loop. For enterprise cloud agents, operation selection is especially challenging because of massive action spaces; hiding it behind a black-box specialist obscures errors and makes inspection and recovery harder. Our work studies where operation-selection decisions should reside within a production architecture.

Tool use, function calling, and API selection. Autonomous tool interaction has rapidly matured. Toolformer demonstrated self-supervised API integration (Schick et al., 2023), API-Bank introduced multi-turn tool-dialogue benchmarks (Li et al., 2023), ToolBench scaled this setting to thousands of real-world REST endpoints with automated retrieval layers (Qin et al., 2024; Guo et al., 2024), and Gorilla used retrieval-aware training to generate accurate API syntax across large collections (Patil et al., 2024). While structured function calling works well for small, static tool sets, enterprise cloud SDKs expose hundreds of service clients and nested schemas; flattening them into context bloats prompts, increases token overhead, and induces hallucinated invocations. Rather than benchmarking native model capacity or synthetic APIs, we analyze whether operation selection should be flattened into context, encapsulated in a subagent, or delegated to deterministic discovery and schema validation. We show that this boundary design is a first-order determinant of agent success and economic efficiency. Recent work also examines tool-interface optimization and interactive evaluation. JTPRO jointly refines agent instructions and tool-specific schema and argument descriptions using rollout-driven reflection, targeting tool selection and slot filling in large tool inventories (Ghoshal et al., 2026). Complementarily, τ-bench evaluates tool-agent-user interactions under domain policies and measures consistency across repeated trials (Yao et al., 2025). These efforts complement our focus on how agent-control patterns and MCP tool boundaries expose operation retrieval, executable schemas, validation feedback, and invocation results over an enterprise-scale SDK catalog.

Retrieval, grounding, and large action spaces. Retrieval-augmented generation (RAG) uses non-parametric text retrieval to improve context efficiency and factual grounding (Lewis et al., 2020). Our architecture applies this principle to executable tools rather than document passages: instead of prompting with complete SDK definitions, the agent retrieves compact operation dossiers. Unlike document RAG, API grounding requires executable contracts with nested parameters, pagination states, identifier provenance, and cloud authentication boundaries. This mirrors sequential decision-making in large or costly action spaces, where superficially similar operations can have different side effects or financial costs (Liu et al., 2026). Our framework adds a deterministic intermediate layer that bounds the action space, keeps retrieval and strict contract validation external to free-form generation, and preserves visibility for the core planner.

Protocols and production tool boundaries. The emerging MCP standardizes how applications expose tools and resources to LMs through a unified client-server architecture (Karimova and Dadashova, 2025). However, protocol standardization does not specify how domain-specific intelligence should be divided between the tool server and the prompt. Recent studies likewise emphasize that real-world agent reliability depends heavily on the infrastructure-level execution harness, not only the raw model backbone (Li et al., 2026). Production environments add operational constraints such as evolving schemas, runtime multi-tenancy, token limits, and strict debugging attribution requirements. We evaluate these factors directly by tracking tool errors, total LLM calls, token usage, and operational latency alongside final task completion.