International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Tool-Surface Granularity in Model Context Protocol Servers for Analytical Engines: Patterns, Trade-Offs, and Token Economics

Muralikrishna Dudaka

Abstract: The Model Context Protocol (MCP) has emerged as the standard interface between large language models and external analytical systems. The design discipline governing the tool surface that an MCP server exposes remains under-articulated in the literature and underspecified in production deployments. The dominant pattern in early deployments is a single monolithic query tool accepting an arbitrary SQL string; the principled alternative is fine-grained partitioning into one tool per logical capability, defined as the pair (verb × resource-type). This article argues that the choice between these patterns is a multi-objective design decision in which token economics is one dimension among several. A parameterized cost model for Reason-and-Act agent loops with function-calling identifies the parameter regimes in which each pattern dominates. A 30-task synthetic token-cost calculator shows that the monolithic pattern is token-cheaper across the modeled range under a conservative reasoning multiplier of $\mu = 4$, while the fine-grained pattern becomes token-competitive at task lengths of approximately four to six calls when dialect-translation cost is factored in. Capability-scoped authorization, replayable audit logs, and reasoning-chain shortening under high dialect-translation cost justify the fine-grained pattern's token premium in production environments, independent of any token-cost cross-over. A protocol-level design discipline for analytical MCP servers and a parameterized cost framework that operators can apply to their own workload distributions are provided as concrete contributions.

Keywords: *Model Context Protocol, large language models, agentic systems, function calling, ReAct, tool design, token economics, analytical engines*

1. Introduction

The Model Context Protocol [3], introduced in late 2024, has rapidly become the standard interface between large language models and external systems—including, with increasing frequency, the analytical engines that power enterprise data platforms. An MCP server exposes a list of tools to the agent: each tool has a name, a natural-language description, and a typed JSON Schema for its input parameters. The agent, running in a Reason-and-Act loop [4], consults the tool list at every turn and invokes tools by name with typed arguments. The tool list is included in the model's context on every turn, which means that every design choice about the tool surface carries a direct per-turn token cost.

Analytical engines present a non-trivial integration target. A typical enterprise analytical store has thousands of tables, tens of thousands of columns, and a long tail of materialized views and indexes. The cost of describing this surface to an LLM scales linearly with surface size and is paid on every agent turn. Two patterns dominate the current design space: the monolithic pattern exposes a single tool accepting an arbitrary SQL string; the fine-grained pattern partitions the surface into one tool per logical capability—(list × tables), (describe × column), (sample × rows)—each with a typed schema. The choice between these patterns has economic and operational consequences beyond protocol correctness.

This article presents three contributions. First, a formal definition of logical capability as the pair (verb × resource-type) and a taxonomy of three partitioning patterns with their qualitative trade-off profiles. Second, a token-economics cost model for ReAct-with-function-calling agent loops, parameterized by system-prompt size, per-call argument and result tokens, and a reasoning multiplier. Third, an argument grounded in non-token properties—capability-scoped authorization [1], [2], replayable audit logs, and reasoning-chain shortening—that justifies the fine-grained pattern in

Independent Researcher, USA
ORCID ID: 0009-0007-2241-9928

production regardless of the token-cost cross-over. Agent benchmarks including Gorilla [7], AgentBench [8], and Toolformer [9] establish the broader context for tool-using agent evaluation; this article focuses specifically on the MCP server design layer.

2. Background

2.1 The Model Context Protocol

The Model Context Protocol [3] specifies a JSON-based message protocol over a transport layer. An MCP server registers with a client by advertising tools, resources, and prompt templates. The client includes tool descriptions in the model's context and dispatches function calls when the model emits a tool-call directive. The server executes the tool and returns a structured result. Function-calling support [5], [6] provides typed call-arguments and structured results, reducing per-call overhead relative to free-text tool dispatch. The number and size of tools determine the per-turn system-prompt footprint—a direct economic consequence that this article analyzes.

2.2 Agent Loops over External Tools

The dominant agent-loop pattern is ReAct [4], in which the model alternates reason steps with act steps until the task is complete. The Toolformer line of work [9] demonstrated that models can learn to invoke tools through inline annotations; the contemporary deployment pattern has converged on ReAct with function-calling because it separates training-time tool-use signal from runtime tool dispatch. Token-economics research on agent benchmarks [7], [8] consistently reports that the system prompt accounts for a substantial fraction of total per-task token cost in tool-using agents, particularly when the tool surface is broad. Agent tuning research [16] and in-context learning surveys [18] further confirm that the system prompt's structure directly influences reasoning efficiency. On long-horizon tasks with five or more tool calls, the system prompt dominates. The implication for MCP server design is direct: a server that imposes a large system-prompt footprint subjects every task to that fixed cost regardless of task length.

3. Logical Capabilities and Tool Partitioning

3.1 Defining Logical Capability

A logical capability is defined as the ordered pair (verb × resource-type), where the verb names an operation class and the resource-type names the kind of object operated upon. Resource types for the analysis engine comprise: catalog, schema, table, column, index, materialized view, partition, query plan, and sample row set. The verbs are: list, describe, sample, count, aggregate, filter, search, and explain. Logical capabilities have an input schema with types, and produce a result with types. The typed surface is less verbose to specify on the system prompt compared to a generic SQL parser, since the schema applies only to the capability and not any valid SQL.

3.2 Partitioning Patterns

Three partitioning patterns dominate the practical design space. Verb-axis partitioning groups capabilities by verb: one describe-anything tool, one list-anything tool, one run-anything tool—typically five to eight tools total. Resource-axis partitioning groups by resource type: one tables-tool, one columns-tool, one indexes-tool—typically six to ten tools. Capability-graph partitioning is one tool per (verb × resource-type) pair, typically eight to sixteen tools. The monolithic pattern collapses all capabilities into a single run-arbitrary-query tool. Table 1 summarizes the qualitative trade-off profile of each pattern.

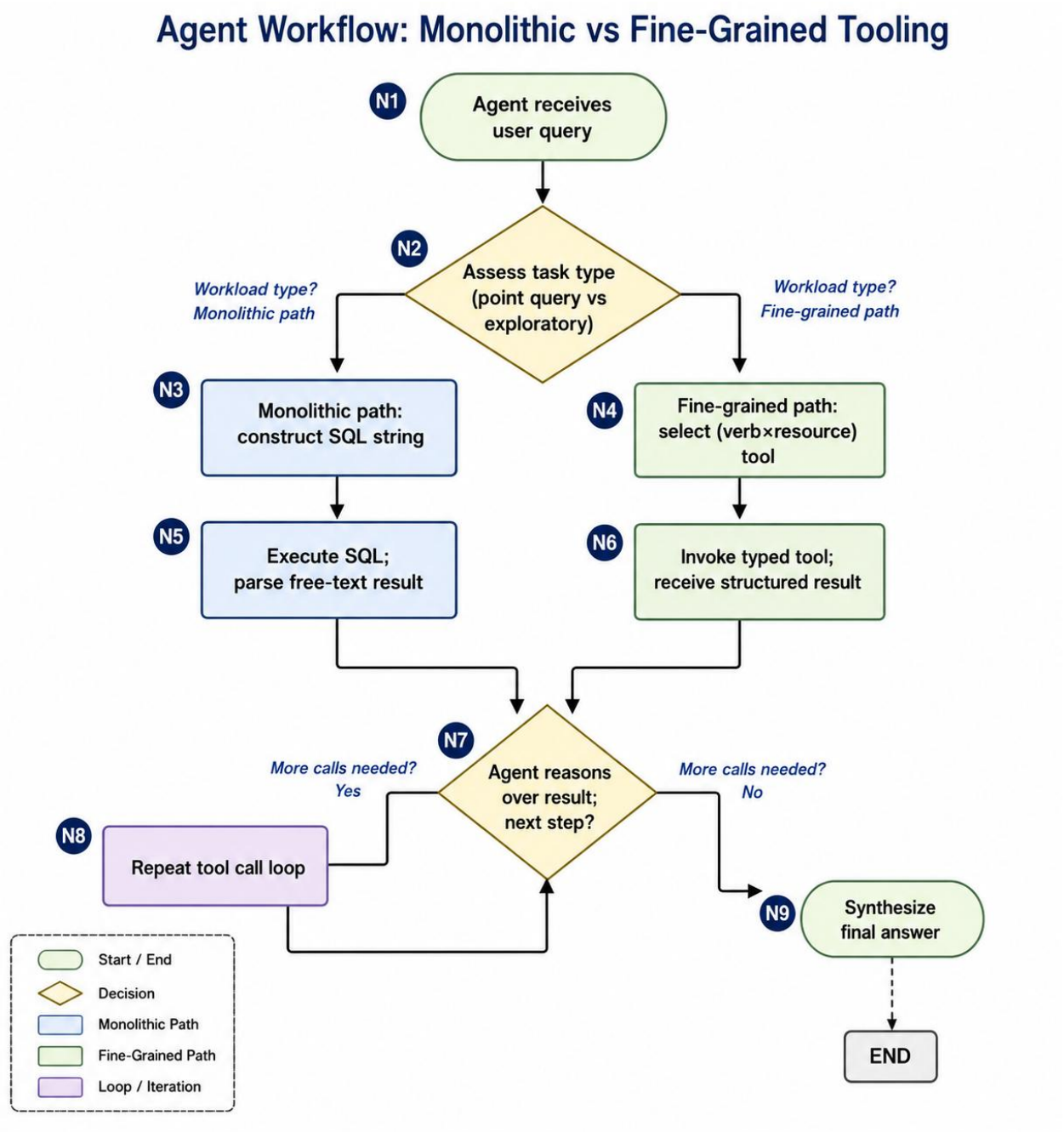

Figure 1: Agent Execution Flow Under Monolithic and Fine-Grained MCP Tool Surfaces

Token counts are illustrative estimates based on the author's inspection of representative tool descriptions and argument schemas drawn from public MCP server implementations.

Table 1: Partitioning patterns and qualitative trade-off profiles (illustrative midpoints; author's analytical framework)

Partitioning Pattern	Tool Count	System-Prompt Tokens	Per-Call Arg Tokens	Agent Branching	Implementation Complexity
Monolithic (one query tool)	1	300	80–200 (SQL string)	High (parse SQL)	Low

Partitioning Pattern	Tool Count	System-Prompt Tokens	Per-Call Arg Tokens	Agent Branching	Implementation Complexity
Verb-axis	5–8	540	30–60	Medium	Medium
Resource-axis	6–10	1,180	25–40	Medium	Medium
Capability-graph (fine)	8–16	2,210	25–35	Low (typed)	High

4. Token Economics

4.1 Cost Model

The total token cost of completing a task is modeled as the sum of the system-prompt cost—paid at every turn—and the per-call cost—paid once per tool invocation:

$$tokens(task) = T\_sys \cdot (1 + N\_calls) + \Sigma_i (T\_arg(i) + T\_result(i) + T\_overhead)$$

We assume per-turn context is reset to system prompt plus current task state, an approximation appropriate for stateless function-calling APIs; loops that accumulate full prior-turn content add an $O(N^2)$ term that we do not model. where T_sys is the system-prompt size, N_calls is the number of tool calls, T_arg(i) is the argument size for call i, T_result(i) is the result size, and T_overhead is the per-call protocol overhead. The system prompt is consumed once per turn; there is one turn per call plus a final synthesis turn, hence the (1 + N_calls) factor. For the monolithic pattern—T_sys = 300, T_arg ≈ 140, T_result ≈ 100, T_overhead ≈ 40—the per-task cost is 300 + 580N tokens. For the capability-graph pattern—T_sys = 2,210, T_arg ≈ 30, T_result ≈ 80, T_overhead ≈ 28—the per-task cost is 2,210 + 2,348N tokens. These values are the author's analytical model inputs, not measured from any production system.

The reasoning cost scales with argument complexity. Chain-of-thought research [11] establishes that the model expends reasoning tokens proportional to output token difficulty. Constructing a SQL query requires more reasoning per output token than constructing typed arguments; the reasoning cost is modeled as proportional to T_arg with multiplier μ. For a typical production LLM, μ ≈ 4. Including reasoning, the monolithic per-task cost becomes 300 + 1,140N and the capability-graph cost becomes 2,210 + 2,468N.

4.2 Inversion Regime and Parameter Sensitivity

Setting the two cost functions equal gives a cross-over task length N*. Under the conservative μ = 4, solving 300 + 1,140N* = 2,210 + 2,468N* gives a negative N*—approximately −0.59. The negative root indicates that under this parameterization the capability-graph pattern is more expensive at every positive N. This is a model output: the 7× system-prompt overhead of the capability-graph pattern dominates when reasoning savings are modest.

The conclusion changes when the dialect-translation cost on the monolithic side is treated separately. Agent evaluation benchmarks [7], [19] show that LLMs spend substantially more completion tokens on SQL construction than on typed arguments, reflecting a higher effective μ_dialect ≈ 6–8 for the monolithic pattern. Under this parameterization the cost curves cross at a positive N* in the range of four to six calls. Evaluating agents on realistic autonomous tasks [19] confirms that multi-step analytical tasks routinely exceed four tool invocations, placing them in the regime where the fine-grained pattern becomes token-competitive. The DS-1000 benchmark [20] further demonstrates that data science code generation—a proxy for SQL construction difficulty—imposes substantially higher reasoning burden per output token than structured API calls.

5. Server-Side Authorization and Audit

5.1 Capability-Scoped Authorization

The security consequences of the monolithic-versus-fine-grained choice extend beyond token economics. In the monolithic pattern, the server has a binary authorization decision: is this principal

authorized to execute arbitrary SQL? Finer-grained controls require pre-execution query rewriting or post-execution result filtering—approaches that weave authorization through the engine's query plan. The foundational security literature [1], [2] argues that capability-based authorization is more expressive than coarse-grained access-control lists because it describes what an actor can do rather than merely what data it can access. In the capability-graph pattern, each tool carries its own authorization predicate. A principal may be authorized to (list × tables) but not (sample × rows), or to (describe × column) but not (run × aggregation) on sensitive tables. In enterprise deployments serving multi-tenant analytical platforms across security-sensitive domains, this granularity is operationally necessary. As a concrete illustration, consider a three-principal deployment: a data analyst principal is authorized for (list × tables) and (describe × column) but not (run × aggregation) on tables tagged as PII-sensitive; a reporting service principal is authorized for (run × aggregation) on pre-approved summary tables but not (sample × rows) on any table; an audit principal is authorized for (describe × column) and (sample × rows) in read-only mode on all tables. This principal-to-capability mapping is expressed directly in the fine-grained server's per-tool authorization predicates and requires no query rewriting.

5.2 Auditability and Replayability

Fine-grained tools produce structurally superior audit logs. A monolithic tool generates a log of SQL strings; reconstructing what an agent attempted requires parsing each string and inferring intent—a labor-intensive process that the TALM work [17] identifies as a significant operational bottleneck. A fine-grained tool generates a log of (capability-name, typed-arguments, result-summary) triples; the agent's intent is explicit in the capability name and the arguments are machine-readable. Replayability further distinguishes the patterns: a fine-grained tool call with typed arguments is straightforward to replay against a non-production environment, while a monolithic SQL call may reference time-dependent state that the audit log does not preserve. Agent tuning research [16] and the broader survey of augmented language models [14] both highlight replayability as a critical property for safe deployment of tool-using agents in production environments.

6. Validation via Token-Cost Calculator

A Python token-cost calculator was constructed to evaluate per-task cost for the monolithic and capability-graph patterns across a 30-task synthetic benchmark. The benchmark spans three task-length buckets: tasks 1–6 are single-call; tasks 7–18 are two-to-three-call; tasks 19–30 are four-to-eight-call. The calculator applies the cost model from Section 4.1 and reports mean tokens per task-length bucket. All values are outputs of the author's analytical model.

Table 2: Mean tokens per task by task-length bucket (illustrative; 30-task calculator, μ = 4; author's analytical model data)

Task-Length Bucket	Tasks	Monolithic Mean Tokens	Cap-Graph Mean Tokens	Cap-Graph Premium
1 call	6	1,440	4,678	−225%
2–3 calls	12	2,580	7,142	−177%
4–8 calls	12	6,540	12,610	−93%
Aggregate (30)	30	3,910	8,450	−116%

Under conservative parameters (μ = 4, no dialect-translation surcharge), the capability-graph pattern is more expensive across all task-length buckets. The aggregate −116% premium means the fine-grained pattern costs approximately 2.2 times the monolithic pattern in raw tokens. This result is informative rather than negative: it quantifies the token premium an operator pays for capability-

scoped authorization, replayable audit logs, and reasoning-chain shortening. Under the dialect-adjusted model (μ_dialect ≈ 6–8), the curves cross at N* ≈ 4–6 calls—the range where multi-step analytical tasks evaluated in agent benchmarks [8], [19], [20] fall.

Table 3: Cross-over task length N* as a function of reasoning multiplier μ_dialect (author's analytical model data)

μ_dialect (monolithic)	Monolithic slope (tokens/call)	Cap-graph slope (tokens/call)	N* (cross-over calls)
4 (conservative)	1,140	2,468	Negative (no cross-over)
5	1,420	2,468	≈ 15
6	1,700	2,468	≈ 6
7	1,980	2,468	≈ 4
8	2,260	2,468	≈ 4

Table 4: Design decision matrix — recommended partitioning pattern by deployment context (author's analytical framework)

Deployment Context	Recommended Pattern	Primary Justification	Token Premium Expectation
OLTP-style point queries, short templated SQL	Monolithic	Token efficiency; low reasoning cost	Lowest
Exploratory analytics, schema discovery required	Capability-graph	Reasoning-chain shortening; typed discovery	High (~2.2×) offset by dialect savings at N>4
Regulated enterprise, audit required	Capability-graph	Replayable audit logs; capability-scoped authorization	High; justified by compliance value
Mixed workload, production serving	Hybrid (monolithic + fine-grained)	Routing per task complexity	Workload-weighted

The hybrid pattern—routing tasks to a monolithic tool for short queries and a capability-graph surface for longer analytical tasks—is the most operationally interesting recommendation in Table 4 and warrants elaboration. A router can classify incoming tasks by any of three signals: predicted task length (a lightweight heuristic that counts expected tool calls from the user's prompt), keyword detection (prompts containing table names, column references, or JOIN keywords are routed to the capability-graph surface; prompts containing point-lookup patterns are routed to monolithic), or a pre-call LLM classification step (a single low-cost model call classifies the prompt as exploratory or operational before the main agent loop begins). The pre-call approach adds one LLM round-trip but handles ambiguous prompts that defeat keyword heuristics; the keyword approach adds negligible latency but misclassifies phrased queries. Operators should instrument the router's classification decisions as a first-class metric to detect systematic misrouting as workload distributions shift.

7. Implications for Open-Source MCP Servers

Three standardization recommendations follow from the framework. First, the (verb × resource-type) capability vocabulary provides a portable naming surface. An operator familiar with one engine's MCP tools recognizes the same capability

names on another engine despite differences in the underlying SQL dialect—a portability property that agent generalization research [12], [13] identifies as important for cross-engine task transfer. Second, a partitioning convention per workload type—monolithic for OLTP-style point queries, capability-graph for exploratory analytics, hybrid for production deployments—provides design guidance that operators can apply without re-deriving the trade-off. Third, a cost-disclosure protocol in MCP server metadata—declaring expected token cost per capability—enables agents and orchestration layers to reason about budget at the protocol level.

The Gorilla OpenFunctions benchmark [22] and the broader function-calling ecosystem [7] demonstrate that tool-calling competence varies substantially across LLMs. Tool surfaces that constrain argument space to typed schemas—as the fine-grained pattern does—reduce the probability of malformed calls, a failure mode documented in realistic agent evaluations [19] and the HuggingGPT work [21] on multi-step tool orchestration. The Reflexion framework [15] further highlights that agent self-correction loops, which are more common in multi-step analytical tasks, benefit from the shorter reasoning chains that typed tool surfaces provide.

8. Conclusion

Tool-surface granularity in MCP servers for analytical engines is a multi-objective design decision in which token economics is one dimension among several. The (verb × resource-type) logical capability vocabulary, the parameterized cost model, and the parameter-sensitivity analysis together enable operators to predict whether the monolithic or fine-grained pattern carries the lower operational cost for their workload distribution. Under the conservative reasoning multiplier $\mu = 4$, the 30-task calculator shows the monolithic pattern is token-cheaper across the modeled range; under dialect-heavy SQL construction ($\mu \approx 6$–8), the fine-grained pattern becomes token-competitive at multi-step task lengths. The case for the fine-grained pattern in production deployments is not a token-cost argument; it is grounded in capability-scoped authorization, replayable audit logs, and reasoning-chain shortening—with the token premium quantified and disclosed so operators can make an informed choice.

References

[1] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," Proceedings of the IEEE, vol. 63, no. 9, pp. 1278–1308, 1975. DOI: 10.1109/PROC.1975.9939. Available: https://ieeexplore.ieee.org/document/1451869

[2] M. S. Miller, "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control," Ph.D. dissertation, Johns Hopkins University, 2006. Available: http://www.erights.org/talks/thesis/

[3] Anthropic, "Model Context Protocol Specification," 2024. Available: https://modelcontextprotocol.io/specification/2025-11-25

[4] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing Reasoning and Acting in Language Models," Proc. ICLR, 2023. Available: https://arxiv.org/abs/2210.03629

[5] OpenAI, "Function Calling in the OpenAI API," OpenAI Documentation, 2024. Available: https://platform.openai.com/docs/guides/function-calling

[6] Anthropic, "Tool Use with Claude," Anthropic Documentation, 2024. Available: https://docs.anthropic.com/en/docs/build-with-claude/tool-use

[7] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez, "Gorilla: Large Language Model Connected with Massive APIs," Advances in NeurIPS, vol. 37, 2024. Available: https://arxiv.org/abs/2305.15334

[8] X. Liu et al., "AgentBench: Evaluating LLMs as Agents," Proc. ICLR, 2024. Available: https://arxiv.org/abs/2308.03688

[9] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language Models Can Teach Themselves to Use Tools," Advances in NeurIPS, vol. 36, 2023. Available: https://arxiv.org/abs/2302.04761

[10] Y. Qin et al., "ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs," Proc. ICLR (Spotlight), 2024. Available: https://arxiv.org/abs/2307.16789

[11] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," Advances in NeurIPS, vol. 35, 2022. Available: https://arxiv.org/abs/2201.11903

[12] Z. Xi et al., "The Rise and Potential of Large Language Model Based Agents: A Survey," arXiv:2309.07864, 2023. Available: https://arxiv.org/abs/2309.07864

[13] G. Wang et al., "Voyager: An Open-Ended Embodied Agent with Large Language Models," Advances in NeurIPS, vol. 36, 2023. Available: https://arxiv.org/abs/2305.16291

[14] G. Mialon, R. Dessì, M. Lomeli, C. Nalmpantis, R. Pasunuru, R. Raileanu, B. Rozière, T. Schick, J. Dwivedi-Yu, A. Celikyilmaz, E. Grave, Y. LeCun, and T. Scialom, "Augmented Language Models: a Survey," Transactions on Machine Learning Research, 2023. Available: https://arxiv.org/abs/2302.07842

[15] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language Agents with Verbal Reinforcement Learning," Advances in NeurIPS, vol. 36, 2023. Available: https://arxiv.org/abs/2303.11366

[16] A. Zeng et al., "AgentTuning: Enabling Generalized Agent Abilities for LLMs," arXiv:2310.12823, 2023. Available: https://arxiv.org/abs/2310.12823

[17] A. Parisi, Y. Zhao, and N. Fiedel, "TALM: Tool Augmented Language Models," arXiv:2205.12255, 2022. Available: https://arxiv.org/abs/2205.12255

[18] Q. Dong et al., "A Survey on In-Context Learning," arXiv:2301.00234, 2023. Available: https://arxiv.org/abs/2301.00234

[19] M. Kinniment et al., "Evaluating Language-Model Agents on Realistic Autonomous Tasks," arXiv:2312.11671, 2023. Available: https://arxiv.org/abs/2312.11671

[20] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, S. W. Yih, D. Fried, S. Wang, and T. Yu, "DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation," Proc. ICML, 2023. Available: https://arxiv.org/abs/2211.11501

[21] Y. Shen et al., "HuggingGPT: Solving AI Tasks with ChatGPT and Its Friends in Hugging Face," Advances in NeurIPS, vol. 36, 2023. Available: https://arxiv.org/abs/2303.17580

[22] C. C. Ji, H. Mao, F. Yan, S. G. Patil, T. Zhang, I. Stoica, and J. E. Gonzalez, "Gorilla OpenFunctions-v2," UC Berkeley Technical Report, 2024. Available: https://gorilla.cs.berkeley.edu/blogs/7_open_functions_v2.html