ProMCP: Profiling Token Flows and Latency Costs in Model Context Protocol–Based LLM Agents

Anonymous ACL submission

Abstract

The Model Context Protocol (MCP) aims to standardize the integration of Large Language Models (LLMs) with external tools, yet existing research primarily evaluates functional capabilities while treating the underlying protocol as an opaque black box. This oversight obscures critical inefficiencies in token flows and latency distributed across MCP's decoupled Host-Client-Server architecture. In this paper, we introduce ***ProMCP***, an end-to-end ***pro***filing and instrumentation framework that decomposes the ***MCP*** workflow into a six-stage communication pipeline, enabling granular attribution of computational costs. We evaluate widely varying deployment topologies, from local LLM models to commercial off-the-shelf (OTS) clients—across 20 servers and 169 tools from MCP-Bench and MCP-Universe. Our analysis reveals a distinct inversion in performance bottlenecks: topologies with customized clients devote 56–72% of total tokens and 60–67% of latency to planning and schema injection, whereas OTS clients concentrate over 85% of latency in final answer synthesis. Crucially, actual tool execution constitutes a negligible fraction of the total cost across all configurations. These findings establish a quantitative baseline for protocol overhead and demonstrate that future optimization must target schema orchestration and transport efficiency rather than tool execution speed. The code is available at: https://anonymous.4open.science/r/mcp-F16B.

1 Introduction

Large Language Models (LLMs) are increasingly used as *agents* that interact with external tools and data sources to solve tasks that require fresh knowledge, precise computation, or action in a real environment. Yet, in most deployed systems, connecting an LLM to tools remains largely ad hoc: each model–tool pair often requires bespoke adapters, prompt templates, and runtime glue code (Yang et al., 2024; Qin et al., 2023; Zhuang et al., 2023; Attouche et al., 2024). This fragmentation complicates development, hinders reproducibility, and makes it difficult to compare agent behaviors across systems.

The Model Context Protocol (MCP)[1] is an emerging lightweight, open-source protocol that aims to standardize how LLM applications expose and consume external *tools*, *resources*, and *prompts*. By providing a consistent communication interface between LLM hosts and tool servers, MCP reduces the "$n \times m$" integration problem in which n different LLM backends must be custom-wired to m different tools or services. In principle, this standardization enables interoperable LLM agent ecosystems: tools can be added, swapped, or updated without re-engineering the entire agent stack.

Motivated by MCP's promise of scalability and interoperability, recent efforts have primarily focused on (i) implementing MCP servers and demonstrating tool-calling capabilities, (ii) building agent frameworks that rely on MCP-like tool-use patterns (Yang et al., 2023), and (iii) evaluating tool-augmented agents using answer-level benchmarks (Patil et al.). For instance, MCP-Bench (Wang et al., 2025) and MCP-Universe (Luo et al., 2025) measure success, grounding, and task complexity; TOOLSANDBOX (Lu et al., 2025) studies stateful interactions and interactive tool use; and MCIP (Jing et al., 2025) investigates protocol-level safety considerations. While these benchmarks and analyses are essential for measuring *capability* and *robustness*, they typically treat MCP as an end-to-end black box.

In practice, however, *efficiency*—especially token consumption and latency—is a first-order concern for time-critical applications and cost-sensitive deployments. MCP introduces additional communication steps and context management op-

[1] https://modelcontextprotocol.io/docs

erations (e.g., tool schema discovery and injection), which can substantially affect end-to-end latency and token usage even when the final answer quality is unchanged. Without a detailed understanding of how tokens and time are spent *inside* an MCP workflow, it is difficult to diagnose bottlenecks, compare deployment choices, or design optimizations that target the true sources of overhead.

Profiling token flows and latency costs in MCP-based agents is challenging for several reasons. First, state and execution are distributed across multiple entities (Host, Client, and Server), often separated by process boundaries and transport mechanisms (e.g., STDIO), which complicates end-to-end attribution. Second, MCP workflows may include *hidden* protocol work before a user query is processed: for example, a host may fetch tool schemas from servers and inject them into the LLM prompt, making it non-trivial to separate schema-related tokens from user-provided content. Third, latency is composed of asynchronous hops across components, and agent execution frequently involves multi-turn internal loops (planning, tool selection, retries) that are invisible to end users but expensive in both tokens and time. Finally, real deployments vary widely: LLMs may run locally or in the cloud, and MCP clients may be customized or off-the-shelf, leading to different bottlenecks even under the same workload.

This paper addresses these gaps by providing a systematic, stage-wise measurement of MCP efficiency. We introduce *ProMCP* (*Pro*filing *MCP*), an end-to-end instrumentation framework that correlates timestamps, payloads, and token accounting across the Host–Client–Server boundary. Designed to enable transparent and reproducible evaluation, *ProMCP* records every MCP message and its associated metadata, allowing us to quantify token usage and latency at fine granularity. We evaluate three representative deployment topologies defined by LLM location and client programmability: *local LLM + customized client (L-Cust)*, *cloud LLM + customized client (C-Cust)*, and *cloud LLM + off-the-shelf client (C-OTS)*. For each topology, we decompose the MCP workflow into six stages spanning the path from the user's query to the final answer and report token and latency costs per stage.

We validate our framework *ProMCP* on MCP-Bench (Wang et al., 2025) and MCP-Universe (Luo et al., 2025), covering 20 MCP servers and 169 tools. Our measurements show that topologies with customized MCP clients devote 56–72% of total tokens and 60–67% of total latency to LLM planning and schema injection, whereas the off-the-shelf client setting (Claude Desktop) shifts the dominant cost to final answer synthesis, which exceeds 85% of the total latency. Across all configurations, tool execution contributes only a small fraction of the overall cost. These results demonstrate that the primary efficiency bottlenecks in MCP workflows are often *protocol- and orchestration-related* rather than tool execution itself, and that deployment choices can qualitatively change where computation is spent.

In summary, our contributions are:

- We introduce *ProMCP*, an open-source end-to-end profiling and instrumentation framework for MCP workflows that captures every message and quantifies token and latency costs across six stages of execution.
- We provide token-level efficiency analysis over 15+ MCP servers and 150+ tools, establishing a quantitative baseline for protocol-induced overhead and identifying dominant bottlenecks.
- We compare MCP efficiency across three deployment topologies (*L-Cust*, *C-Cust*, and *C-OTS*), showing how LLM placement and client programmability reshape planning cost, schema injection overhead, and end-to-end latency.

2 Related Work

Tool-augmented LLMs and tool use. Research on tool-augmented LLMs has grown rapidly, driven by the need for models to go beyond free-form text generation and interact with external services (Yuan et al., 2025; Qin et al., 2023; Yang et al., 2024; Parisi et al., 2022; Patil et al.). Early methods such as ReAct (Yao et al., 2023) and Toolformer (Schick et al., 2023) showed that LLMs can be prompted or fine-tuned to invoke APIs during problem solving. However, these approaches typically rely on bespoke, system-specific integrations and provide limited visibility into the orchestration costs incurred by tool discovery, schema management, and tool-call coordination.

Agent frameworks and orchestration systems. In parallel, developer-oriented frameworks such as LangChain[2] and Haystack[3] facilitate composing tools into multi-step pipelines and agentic workflows. Likewise, agent-oriented systems such as

[2] https://docs.langchain.com/oss/python/langchain/tools

[3] https://docs.haystack.deepset.ai/docs/tool

AutoGPT (Yang et al., 2023) and BabyAGI (Talebirad and Nadiri, 2023) explore autonomous planning with LLMs as controllers. While these frameworks improve usability and accelerate prototyping, tool invocation is often treated as an opaque subroutine: the underlying communication patterns are neither standardized nor easily inspectable, making it difficult to audit or attribute efficiency costs at the protocol level.

MCP and MCP-based benchmarks. MCP was introduced to address the standardization gap by defining a uniform interface for how LLM hosts, clients, and tool servers communicate. Building on this protocol layer, MCP execution–focused benchmarks such as MCP-Bench (Wang et al., 2025) and MCP-Universe (Luo et al., 2025) evaluate agents on real MCP servers and time-varying environments, exposing failure modes in tool retrieval, schema adherence, and cross-server reasoning. Related benchmarks such as TOOLSANDBOX (Lu et al., 2025; Hsieh et al., 2023) study stateful interactions and conversational tool use. These efforts provide realistic evaluation settings, but they primarily focus on task success and answer-level outcomes rather than *where* token and latency costs arise within the MCP workflow. A complementary line of work examines process quality and capability profiling: MCP-RADAR (Gao et al., 2025) characterizes tool-use behavior along axes such as accuracy and first-error position, but does not inspect protocol overhead or communication-layer bottlenecks.

Safety, governance, and domain-specific MCP systems. Other work explores and monitors MCP safety and governance (Narajala and Habler, 2025; Radosevich and Halloran, 2025). MCIP (Jing et al., 2025) proposes contextual-integrity tracking and a Guardian model to detect unsafe or inappropriate tool calls, and provides a structured taxonomy of MCP risks. Domain-specific MCP systems further demonstrate extensibility; for example, SensorMCP (Guo et al., 2025) supports automatic tool generation and co-evolving language assets for specialized sensing environments. While these studies highlight MCP's flexibility and security implications, they do not provide systematic measurements of communication cost, token usage, or end-to-end latency across different deployment configurations.

Gap: protocol-level efficiency and cost attribution. Across these categories, a common limitation is the absence of protocol-level efficiency analysis. Existing work does not track how tokens, latency, and communication overhead accumulate across the stages of an MCP interaction, nor does it compare how different deployment modes behave under identical workloads. Our work addresses this gap by providing an end-to-end, token-level analysis of MCP communication flows across deployments, revealing previously hidden bottlenecks in schema injection, planning, transport, and final answer synthesis.

3 Methodology

3.1 *ProMCP* Overview

ProMCP focuses on two complementary metrics: **token cost** and **latency**. Token cost captures context and inference budget pressure, while latency captures the end-to-end delay experienced by users. Measuring both is essential because MCP can increase tokens (through schema and tool-result injection) and also increase time (through additional protocol hops), and these effects may vary substantially across deployment settings.

System overview. Figure 1 illustrates the architecture we profile. An MCP **Host** runs the agent loop around an LLM (local or cloud). The MCP **Client** bridges between the Host and one or more MCP **Servers** that expose tools and resources (e.g., APIs, databases, file systems, web). The client validates the LLM's tool plans, serializes them into JSON-RPC requests, and exchanges responses with servers over transports such as STDIO or HTTP/SSE. *ProMCP* instruments these interfaces with *Token Tracking Module* and *Latency Monitoring Module* (see Figure 1) to record what is communicated, when it is communicated, and how much token budget it consumes when inserted into the LLM context.

Six-stage decomposition. To enable fine-grained attribution, *ProMCP* models each tool-augmented interaction as a six-stage communication pipeline (stages correspond to the numbered markers in Figure 1):

S1 User → LLM (Prompting): the user query is submitted to the Host and forwarded to the LLM.

S2 LLM → Client (Planning): the LLM produces a tool plan (e.g., a tool name with structured arguments).

S3 Client → Server (Tool Call): the client validates the plan, formats a JSON-RPC request, and issues the tool call.

S4 Server → Client (Tool Response): the server executes the tool and returns the result payload to the client.

S5 Client → LLM (Context Update): the client packages tool results (and, when required, schema/context snippets) back into the LLM input.

S6 LLM → User (Answer Synthesis): the LLM generates the final user-facing response.

For each stage, *ProMCP* records (i) token footprint (input/output/total tokens when applicable) and (ii) stage latency. In addition to per-stage reporting, we also aggregate latency into three user-meaningful segments: **tool-plan latency** (S1–S2), **tool-execution latency** (S3–S4), and **answer-synthesis latency** (S5–S6). This staged view enables direct comparisons across deployment topologies under identical workloads.

3.2 Token and Latency Instrumentation

Unified event log. *ProMCP* performs protocol tracing beyond LLM-generated text to capture MCP's representational and transport overhead. For every event in the six-stage pipeline (and for initialization events described in Section 3.3), we write a structured log record containing:

- **Identifiers:** `run_id`, `task_id`, stage index (S1–S6), a semantic phase label (e.g., `llm_plan`, `tool_call`, `final_answer`), and the communication direction (e.g., `user`→`llm`).
- **Timing:** high-resolution timestamps for send/receive boundaries and the derived stage latency (milliseconds). When supported, we additionally record provider-side timing metadata (e.g., model processing time) to separate inference time from transport overhead.
- **Model/tool metadata:** model name, server identifier, tool name, and transport type (STDIO vs. HTTP/SSE), when applicable.
- **Token accounting:** input, output, and total token counts for LLM calls; token *footprint* estimates for payloads that are injected into the LLM context (e.g., schemas, tool results).
- **Payload previews:** truncated request/response excerpts to facilitate qualitative inspection of prompts, schemas, and tool outputs.[4]

[4] In our released logs, sensitive fields can be redacted or hashed; the token and latency accounting remains unchanged.

Representative Token-Level Log Entry


```
{
  "run_id": "claude-sonnet-4.5_wx_current...",
  "task_id": "wx_current_springfield_nz",
  "stage": "S6",
  "phase": "final_answer",
  "dir": "llm->user",
  "timestamp": "2025-11-25T05:33:14Z",
  "latency_ms": 6061.06,
  "model": "claude-sonnet-4.5",
  "tokens_in": 555,
  "tokens_out": 321,
  "tokens_total": 876,
  "req_preview": "...",
  "resp_preview": "..."
}
```


Token footprint vs. LLM usage tokens. A key challenge is that MCP introduces large textual artifacts (e.g., JSON Schemas and tool results) whose cost manifests when they are inserted into the LLM input, even though they originate outside the model. *ProMCP* therefore distinguishes: (i) *LLM usage tokens* (returned by the provider or computed by the local tokenizer for prompt/response pairs), and (ii) *protocol token footprint* (tokenized size of MCP artifacts—schemas, JSON tool calls, and tool results—under the same tokenizer used by the corresponding LLM). This separation allows us to attribute costs to protocol representation (e.g., verbose schemas) versus model reasoning and generation.

Latency measurement. We measure latency at stage boundaries using monotonic clocks within the instrumented component(s). For LLM calls (S1/S2/S6), latency is measured from request dispatch to completion (or last streamed chunk). For tool calls (S3/S4), latency is measured from JSON-RPC request emission to receipt of the corresponding response. This makes per-stage delays comparable across STDIO and networked deployments, and highlights where bottlenecks occur (planning, transport, tool execution, or synthesis).

3.3 Handling of *Hidden* Protocol Costs

Before processing any user queries, an MCP client establishes a session with each server. This *initialization* phase typically includes: (i) an `initialize` handshake and metadata exchange, (ii) a `tools/list` request where the server exposes available tools as JSON Schemas, and (iii) a readiness confirmation that the session can accept tool calls. Although no user prompt has been issued, this phase can incur substantial cost: schemas may

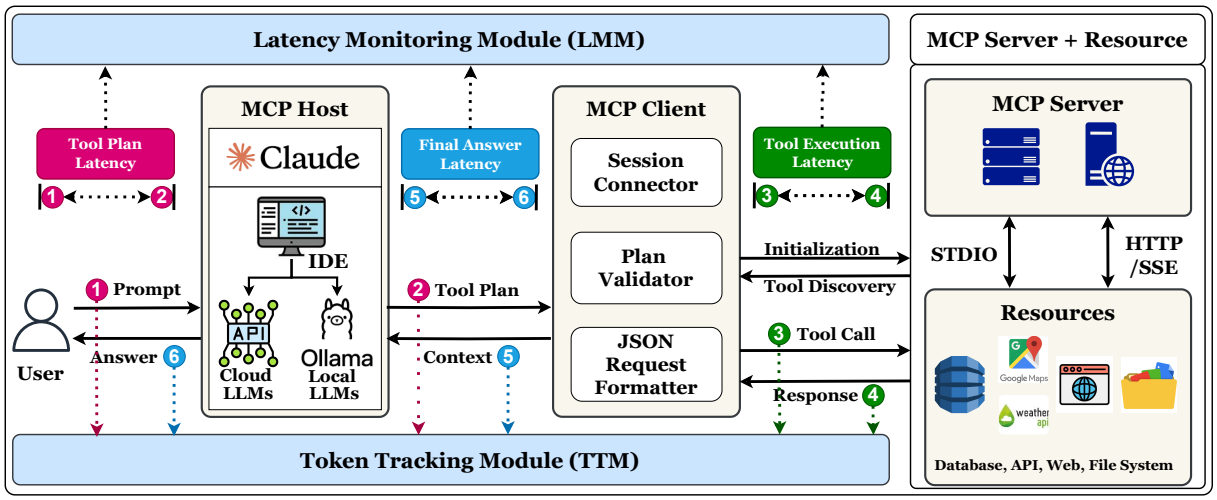

Figure 1: *ProMCP* instruments the MCP Host–Client–Server loop and decomposes each interaction into six stages (S1–S6). The Token Tracking Module (TTM) estimates token footprint of prompts, schemas, tool calls, and tool results (as they are injected into the LLM context), while the Latency Monitoring Module (LMM) measures stage-level delays. We additionally log session initialization and tool discovery (`initialize` and `tools/list`) to capture hidden protocol overhead before user queries.

be large and nested, requiring serialization, transport, parsing, and (in many stacks) injection into the LLM context before planning begins. To prevent this overhead from being silently amortized or ignored, *ProMCP* treats session initialization as a first-class part of the MCP lifecycle and logs both token footprint and latency for the handshake and tool discovery events. We report these costs separately in our analysis to quantify how much is spent *before* tool-augmented reasoning starts.

3.4 MCP Deployment Topologies

To study how deployment choices affect MCP efficiency, we evaluate three representative deployment topologies defined by (i) where the LLM runs (local vs. cloud) and (ii) whether the client is customizable or off-the-shelf:

L-Cust: Local LLM + Customized Client.

This configuration represents a fully local stack: the LLM and the MCP client run on the user's machine, and servers are connected via STDIO. It isolates protocol overhead from cloud-related delays and provides a lower bound on transport latency. This setting is also representative of privacy-preserving or air-gapped deployments.

C-Cust: Cloud LLM + Customized Client.

This hybrid configuration uses a cloud-hosted LLM (inference via remote API) while retaining a locally implemented and modifiable MCP client. Tool schemas and tool results must traverse the network as part of LLM prompting and context updates, capturing combined effects of API request/response packing, network latency, and server-side inference.

C-OTS: Cloud LLM + Off-the-Shelf Client.

This configuration uses a commercial, off-the-shelf MCP client/host stack (e.g., a GUI application). Schema injection, planning, and tool invocation are managed internally by the application. Although the underlying model family may match the API setting, implementation details (prompt orchestration, planning heuristics, internal formatting, buffering/streaming behavior) can substantially reshape token usage and latency. This topology serves as a realistic baseline for end-user MCP deployments.

3.5 Cross-Topology Log Normalization and Aggregation

Directly comparing MCP costs across topologies requires a common representation. *ProMCP* normalizes all collected traces into a unified schema with one record per (task, stage), enabling consistent aggregation and statistical analysis.

L-Cust. For local execution, we obtain token and latency measurements from the customized client's trace hooks, which record all LLM and server messages during STDIO-based execution. Token footprint is computed using the local model's tokenizer so that schema and tool-result injection are measured under the same encoding as the LLM.

C-Cust. For API-based execution, we extract token usage (input/output tokens) and any available model-side timing metadata from the provider responses, then align these with client-side protocol traces using request identifiers and timestamps. This produces an end-to-end view that combines cloud inference costs with local orchestration and transport overhead.

C-OTS. For off-the-shelf clients, protocol-level traces are not directly exposed. We therefore reconstruct the six-stage pipeline from exported conversation logs (e.g., `conversations.json`) by identifying tool-use messages, mapping them to stages S1–S6, computing token counts using the corresponding tokenizer, and deriving per-stage latencies from timestamp deltas. While this reconstruction cannot reveal internal intermediate states beyond the exported data, it enables systematic comparison of end-to-end costs against customized-client settings.

Output format. All topology-specific traces are transformed into parallel CSV/JSON summaries. The resulting dataset provides a unified, stage-aligned view of MCP communication flows across local, API-based, and off-the-shelf deployments, which we analyze in Section 4.3.

4 Evaluations

4.1 Benchmark Suites and Workloads

We ground our experiments in MCP-Bench (Wang et al., 2025) and MCP-Universe (Luo et al., 2025), two benchmark datasets designed to evaluate tool-augmented LLM systems operating over MCP. Unlike earlier synthetic API suites, MCP-Bench constructs tasks directly against real MCP servers and production-grade tools, capturing the architectural and interactional complexity an MCP client must handle in practice. This diversity is critical for stress-testing retrieval and orchestration mechanisms under nonuniform tool metadata and mixed I/O signatures. We extend our evaluation to MCP-Universe (Luo et al., 2025), composed of longer, open-ended user queries that reflect realistic, end-to-end agent workloads. While MCP-Bench emphasizes controlled task structure, MCP-Universe stresses multi-step planning, large tool responses, and sustained context management across complex workflows. For this study, we integrate 20 servers exposing 169 tools spanning research, finance, geospatial analysis, scientific computing, and other verticals.

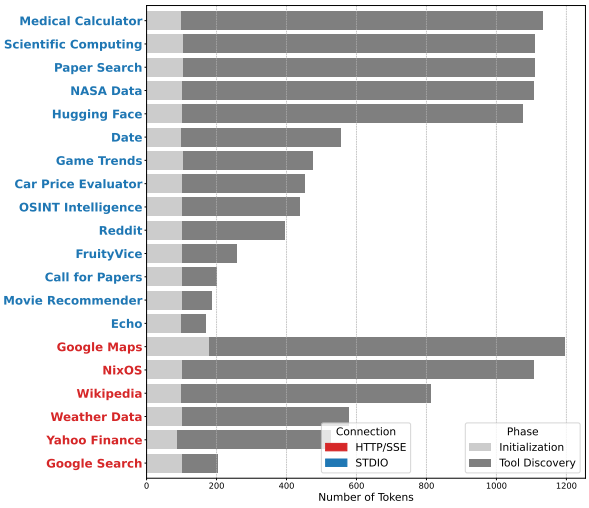

Figure 2: Token distribution during MCP session initialization and tool discovery across 20 servers connected via STDIO and HTTP/SSE.

4.2 Experimental Setup

All experiments were performed on a standalone windows 11 workstation equipped with an NVIDIA GeForce RTX 4090 GPU, CUDA acceleration, an Intel Core i9-13900K CPU, and 64 GB of system memory. We evaluate MCP across three LLM deployment topologies: L-CUST, consisting of locally hosted LLMs served via Ollama[5] and customized MCP clients using FastMCP, including Mistral Small 3.2 24B and LLaMA-3.2; C-CUST, representing cloud-hosted LLMs accessed via the Claude API (Claude Sonnet 4.5) with customized MCP clients using FastMCP; and C-OTS, denoting cloud LLMs instantiated through off-the-shelf Claude Desktop (Claude Sonnet 4.5).

4.3 Experimental Results

4.3.1 Initialization Overhead Analysis

Before an agent can reason, it must discover available tools. Figure 2 illustrates the token cost of this often-overlooked phase. We observe that the `tools_discovery` step dominates initialization costs, as the server must serialize and transmit verbose JSON schemas for every available tool.

Figure 3 highlights the impact of transport protocols on this phase. **HTTP/SSE** connections demonstrate significantly lower initialization latency compared to **STDIO**. This is attributed to the "cold start" penalty of STDIO, which requires spawning a new subprocess for every connection, whereas HTTP servers typically remain "warm"

[5] https://docs.ollama.com/

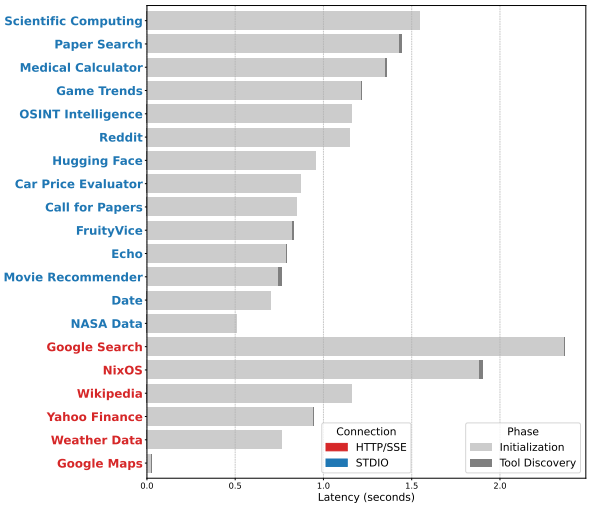

Figure 3: Latency during MCP session initialization and tool discovery across 20 servers connected via STDIO and HTTP/SSE.

and persistent. However, once the connection is established, the `tools_discovery` latency is negligible across both transports, confirming that schema transmission is bandwidth-bound rather than compute-bound.

4.3.2 Token Distribution & Phase Attribution

Table 1 reports the mean and standard deviation of token usage across MCP phases for three execution configurations, C-Cust, C-OTS, and L-Cust (instantiated with both Mistral 3.2 and LLaMA 3.2), on the MCP-Bench and MCP-Universe datasets.

Performance on MCP-Bench. On the MCP-Bench task set, the token profile is primarily defined by how the client handles the fixed overhead of tool definitions. The L-Cust and C-Cust executions exhibit a schema-dominated profile, where the *llm_plan* phase accounts for over half of the total tokens (52–63%). This reflects the baseline structural cost of providing definitions for over 150 tools, which naturally outweighs the concise interactions required for these tasks. In contrast, the C-OTS configuration shows a marked inversion: the planning share drops to 2.1%, drowned out by a massive inflation in the *tool_result* and *final_ans* phases. This suggests that off-the-shelf clients incur significant overhead—likely through unfiltered file reads or verbose context retention—resulting in a total token consumption roughly $4\times$ higher than the custom baselines ($21,050$ vs. $\approx 5,400$).

Performance on MCP-Universe. On the more complex MCP-Universe task set, the profile shifts as execution demands rise. For C-Cust, the dominant cost moves to the *tool_result* phase (77.5%), driven by the heavy retrieval requirements of open-ended queries. L-Cust reveals a significant divergence in model behavior: Mistral 3.2 mirrors the high-volume retrieval pattern of the cloud setup (81.8% tool results), whereas LLaMA 3.2 remains highly compact, with *tool_result* consuming only 20.0% of its budget. The C-OTS configuration exhibits extreme amplification, with a total volume exceeding 1.2 million tokens—orders of magnitude higher than custom setups—demonstrating how unconstrained tool interactions can exponentially inflate costs in realistic agentic workflows.

4.3.3 Latency Profiling & Bottleneck Analysis

Table 2 presents the mean and standard deviation of execution latency (in seconds) across MCP phases. The data compares the three configuration types, C-Cust, C-OTS, and L-Cust, across the MCP-Bench and MCP-Universe datasets.

Input Latency and Planning Overhead. On the simpler MCP-Bench tasks, local execution (L-Cust) demonstrates a massive speed advantage. Mistral 3.2 achieves a near-instantaneous total latency of 1.28s, with the planning phase taking less than a second. However, this advantage is fragile. On the complex MCP-Universe dataset, Mistral's performance degrades catastrophically: its total latency jumps to 73.53s, driven by a massive spike in the *llm_plan* phase (46.84s) accompanied by extreme variance ($\sigma \approx 113$s). This indicates that while small local models are highly efficient for simple tool use, they struggle to process the heavier context windows of open-ended tasks, often "hanging" or processing inefficiently during the planning stage. In contrast, LLaMA 3.2 maintains more predictable scaling, with latency increasing moderately from 3.90s to 41.43s.

Cloud Overhead and the "First Token" Cost. The C-Cust (Claude API) configuration reveals the inherent latency floor of cloud-based inference. Even on simple tasks, it incurs a baseline latency of ≈ 22s, dominated by the *llm_plan* phase (14.60s). This suggests that for custom cloud agents, the primary bottleneck is not tool execution, but the pre-filling and processing of the system prompts and tool definitions. As task complexity increases in MCP-Universe, this planning cost nearly doubles (25.27s), confirming that prompt processing remains the dominant latency contributor for custom cloud setups.

Data	Phase	C-Cust (Claude API)		L-Cust (Mistral 3.2)		L-Cust (LLaMA 3.2)		C-OTS (Claude Deskt.)	
		Tokens ($\mu \pm \sigma$)	**%**	**Tokens** ($\mu \pm \sigma$)	**%**	**Tokens** ($\mu \pm \sigma$)	**%**	**Tokens** ($\mu \pm \sigma$)	**%**
Bench	context	420 ± 95	7.8	400 ± 88	7.6	405 ± 91	6.9	893 ± 87	4.2
	llm_plan	$3{,}411 \pm 318$	63.2	$3{,}029 \pm 269$	57.3	$3{,}026 \pm 398$	52.0	433 ± 32	2.1
	tool_call	19 ± 7	0.4	24 ± 11	0.5	63 ± 16	1.1	445 ± 41	2.1
	tool_result	423 ± 213	7.8	559 ± 348	10.6	$1{,}477 \pm 385$	25.4	$10{,}229 \pm 516$	48.6
	final_ans	$1{,}128 \pm 407$	20.9	$1{,}273 \pm 559$	24.1	$1{,}851 \pm 794$	31.8	$9{,}050 \pm 402$	43.0
	Total	$\mathbf{5{,}401 \pm 1{,}040}$	**100**	$\mathbf{5{,}285 \pm 1{,}275}$	**100**	$\mathbf{5{,}822 \pm 1{,}684}$	**100**	$\mathbf{21{,}050 \pm 1{,}078}$	**100**
Universe	context	$1{,}012 \pm 214$	1.9	900 ± 198	1.4	$1{,}080 \pm 226$	9.6	$1{,}120 \pm 120$	0.1
	llm_plan	$6{,}802 \pm 695$	12.9	$6{,}340 \pm 2{,}723$	9.8	$5{,}686 \pm 991$	50.6	$126{,}805 \pm 214$	9.9
	tool_call	186 ± 139	0.4	188 ± 121	0.3	164 ± 85	1.5	$3{,}761 \pm 526$	0.3
	tool_result	$41{,}001 \pm 64{,}890$	77.5	$53{,}161 \pm 85{,}875$	81.8	$2{,}247 \pm 2{,}494$	20.0	$1{,}063{,}220 \pm 19{,}592$	82.9
	final_ans	$3{,}929 \pm 3{,}070$	7.4	$4{,}370 \pm 2{,}123$	6.7	$2{,}063 \pm 802$	18.3	$87{,}411 \pm 506$	6.8
	Total	$\mathbf{52{,}930 \pm 18{,}204}$	**100**	$\mathbf{64{,}959 \pm 31{,}991}$	**100**	$\mathbf{11{,}240 \pm 3{,}006}$	**100**	$\mathbf{1{,}282{,}317 \pm 4{,}118}$	**100**

Table 1: Token distribution across MCP phases and deployment topologies.

Data	Phase	C-Cust (Claude API)		L-Cust (Mistral 3.2)		L-Cust (LLaMA 3.2)		C-OTS (Claude Deskt.)	
		Lat ($\mu \pm \sigma$, **s**)	**%**	**Lat** ($\mu \pm \sigma$, **s**)	**%**	**Lat** ($\mu \pm \sigma$, **s**)	**%**	**Lat** ($\mu \pm \sigma$, **s**)	**%**
Bench	context	0.42 ± 0.11	1.9	0.10 ± 0.04	10.1	0.11 ± 0.05	2.8	0.20 ± 0.12	0.2
	llm_plan	14.60 ± 2.62	66.5	0.59 ± 0.31	46.1	2.21 ± 1.10	56.7	5.87 ± 1.77	6.2
	tool_call	0.08 ± 0.03	0.3	0.01 ± 0.01	<0.1	0.03 ± 0.02	0.8	7.01 ± 5.30	7.4
	tool_result	0.56 ± 0.55	2.5	0.01 ± 0.01	0.9	0.01 ± 0.01	0.3	0.45 ± 0.78	0.5
	final_answer	7.10 ± 1.77	32.3	0.57 ± 0.44	42.9	1.54 ± 0.79	39.4	81.97 ± 16.34	86.4
	Total Latency	$\mathbf{22.76 \pm 4.21}$	**100**	$\mathbf{1.28 \pm 0.52}$	**100**	$\mathbf{3.90 \pm 1.43}$	**100**	$\mathbf{94.86 \pm 18.11}$	**100**
Universe	context	0.85 ± 0.21	1.9	1.10 ± 0.25	1.5	0.95 ± 0.23	2.3	0.20 ± 0.12	0.2
	llm_plan	25.27 ± 4.12	55.7	46.84 ± 113.38	63.7	22.82 ± 14.05	55.1	5.87 ± 1.77	6.2
	tool_call	0.40 ± 0.14	0.9	0.55 ± 0.20	0.7	0.38 ± 0.08	0.9	7.01 ± 5.30	7.4
	tool_result	3.61 ± 2.56	8.0	3.09 ± 2.11	4.2	1.39 ± 0.81	3.4	0.45 ± 0.78	0.5
	final_answer	16.48 ± 6.25	36.3	23.60 ± 25.34	32.1	17.22 ± 16.34	41.6	81.97 ± 16.34	86.4
	Total Latency	$\mathbf{46.61 \pm 9.18}$	**100**	$\mathbf{73.53 \pm 27.81}$	**100**	$\mathbf{41.43 \pm 18.22}$	**100**	$\mathbf{94.86 \pm 18.11}$	**100**

Table 2: Latency distribution across MCP phases and deployment topologies.

Output Bottlenecks in Off-the-Shelf Clients. The C-OTS configuration exhibits a distinct latency profile that is inversely related to the custom setups. Its total latency is consistently high ($\approx$ 94s) but is driven almost entirely by the *final_ans* phase (81.97s), which accounts for over 86% of the execution time. Unlike custom agents, which are bottlenecked by input processing (planning), the OTS client is bottlenecked by output generation. This correlates with the massive token volume observed in Table 1, confirming that the verbose, streaming response style of the desktop client incurs a severe penalty on user-perceived latency.

Overall, the results highlight a shift in bottlenecks based on implementation. Custom environments (L-Cust, C-Cust) generally suffer from an "Input Bottleneck," where latency is determined by how quickly the model can parse tool definitions and plan. Off-the-shelf environments (C-OTS) suffer from an "Output Bottleneck," where unconstrained generation and streaming time dominate the user experience.

5 Conclusion

We introduce *ProMCP*, an end-to-end profiling framework that demystifies MCP efficiency by decomposing execution into a six-stage pipeline. Evaluating three deployment topologies across 20 servers and 169 tools, we find that protocol orchestration—rather than tool runtime—dominates cost. Specifically, customized clients expend the majority of resources on planning and schema injection, whereas off-the-shelf clients are bottlenecked by answer synthesis. These results demonstrate that future optimizations must prioritize schema management and transport-aware orchestration. By establishing a reproducible baseline for token and latency attribution, *ProMCP* lays the groundwork for designing next-generation, efficiency-optimized MCP agents.

Limitations

While *ProMCP* provides a comprehensive view of MCP efficiency, our study has three primary limitations. First, our analysis of commercial clients (C-OTS) relies on post-hoc log reconstruction from `conversations.json` rather than real-time introspection. This method aggregates total latency but prevents us from measuring precise millisecond-level jitter or internal retries that do not result in a user-visible message. Second, our experiments were conducted on a single hardware configuration (Windows 11 workstation). While we isolated network vs. local latency, variations in OS-level scheduling or STDIO buffer sizes on Linux or macOS could introduce minor performance deviations in the L-Cust topology. Finally, our token analysis uses the tokenizer specific to each model (e.g., Claude or LLaMA). Although this ensures accuracy for each specific deployment, it makes direct cross-model token comparisons approximate due to vocabulary differences. Future work should explore unified metrics for schema density to normalize these comparisons.

References

Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2024. Validation of modern json schema: Formalization and complexity. *Proceedings of the ACM on Programming Languages*, 8(POPL):1451–1481.

Xuanqi Gao, Siyi Xie, Juan Zhai, Shiqing Ma, and Chao Shen. 2025. Mcp-radar: A multi-dimensional benchmark for evaluating tool use capabilities in large language models. *arXiv preprint arXiv:2505.16700*.

Yunqi Guo, Guanyu Zhu, Kaiwei Liu, and Guoliang Xing. 2025. Sensormcp: A model context protocol server for custom sensor tool creation. In *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services*, pages 747–752.

Cheng-Yu Hsieh, Si-An Chen, Chun-Liang Li, Yasuhisa Fujii, Alexander Ratner, Chen-Yu Lee, Ranjay Krishna, and Tomas Pfister. 2023. Tool documentation enables zero-shot tool-usage with large language models. *arXiv preprint arXiv:2308.00675*.

Huihao Jing, Haoran Li, Wenbin Hu, Qi Hu, Xu Heli, Tianshu Chu, Peizhao Hu, and Yangqiu Song. 2025. Mcip: Protecting mcp safety via model contextual integrity protocol. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 1177–1194.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, and 1 others. 2025. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1160–1183.

Ziyang Luo, Zhiqi Shen, Wenzhuo Yang, Zirui Zhao, Prathyusha Jwalapuram, Amrita Saha, Doyen Sahoo, Silvio Savarese, Caiming Xiong, and Junnan Li. 2025. Mcp-universe: Benchmarking large language models with real-world model context protocol servers. *arXiv preprint arXiv:2508.14704*.

Vineeth Sai Narajala and Idan Habler. 2025. Enterprise-grade security for the model context protocol (mcp): Frameworks and mitigation strategies. *arXiv preprint arXiv:2504.08623*.

Aaron Parisi, Yao Zhao, and Noah Fiedel. 2022. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*.

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

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, and 1 others. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.

Brandon Radosevich and John Halloran. 2025. Mcp safety audit: Llms with the model context protocol allow major security exploits. *arXiv preprint arXiv:2504.03767*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools, 2023. *arXiv preprint arXiv:2302.04761*.

Yashar Talebirad and Amirhossein Nadiri. 2023. Multi-agent collaboration: Harnessing the power of intelligent llm agents. *arXiv preprint arXiv:2306.03314*.

Zhenting Wang, Qi Chang, Hemani Patel, Shashank Biju, Cheng-En Wu, Quan Liu, Aolin Ding, Alireza Rezazadeh, Ankit Shah, Yujia Bao, and 1 others. 2025. Mcp-bench: Benchmarking tool-using llm agents with complex real-world tasks via mcp servers. *arXiv preprint arXiv:2508.20453*.

Hui Yang, Sifu Yue, and Yunzhong He. 2023. Auto-gpt for online decision making: Benchmarks and additional opinions. *arXiv preprint arXiv:2306.02224*.

Linyao Yang, Hongyang Chen, Zhao Li, Xiao Ding, and Xindong Wu. 2024. Give us the facts: Enhancing large language models with knowledge graphs for

fact-aware language modeling. *IEEE Transactions on Knowledge and Data Engineering*, 36(7):3091–3110.

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

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Kan Ren, Dongsheng Li, and Deqing Yang. 2025. Easytool: Enhancing llm-based agents with concise tool instruction. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 951–972.

Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. 2023. Toolqa: A dataset for llm question answering with external tools. *Advances in Neural Information Processing Systems*, 36:50117–50143.