Network Working Group
Internet-Draft
Intended status: Informational
Expires: 8 June 2026

Z. Chang
J. Li
Z. Cao
Huawei

5 December 2025

A Token-efficient Data Layer for Agentic Communication
draft-chang-agent-token-efficient-01

Abstract

Agentic communication fundamentally differs from traditional machine communication in that its outputs are recursively consumed and interpreted by Large Language Models (LLMs). This unique characteristic makes efficient use of the model's limited context a critical requirement. However, current agent communication protocols - such as the Model Context Protocol (MCP) and Agent-to-Agent (A2A) - often suffer from token or context bloat, caused by redundant schema definitions, verbose message structures, and inefficient capability exchanges. As a result, a substantial number of tokens are unnecessarily consumed within the model' s context window.

To address these issues, this draft defines the Agentic Data Optimization Layer (ADOL), a general and foundational data-layer for agent communication. ADOL introduces a set of backward-compatible optimizations, including schema deduplication through JSON references, adaptive inclusion of optional fields, controllable response verbosity, and retrieval-based selection mechanisms. Collectively, these mechanisms reduce token consumption, enhance context efficiency, and provide a scalable foundation for future agent communication frameworks.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 8 June 2026.

Copyright Notice

Copyright (c) 2025 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction 2
1.1. Token Bloat Problem in A2A (Agent-Agent Communication) . 3
1.2. Token Bloat Problem in MCP (Agent-Tool Communication) . . 3
1.3. Requirements Language 4
2. Problem Statement 4
2.1. Repetitive Schema Across Tools 4
2.2. Excessive Optional Content 5
2.3. Overly Long Server Responses 5
2.4. Inefficient Tool Selection 6
3. Solution Overview 6
3.1. Schema Deduplication Using JSON References 7
3.2. Adaptive Optional Field Inclusion 8
3.3. Controllable Response Verbosity 8
3.4. Tool Selection Mechanism 9
4. Specification 10
5. IANA Considerations 11
6. Security Considerations 11
7. References 11
7.1. Normative References 11
7.2. Informative References 11
Acknowledgements 12
Authors' Addresses 12

1. Introduction

1.1. Token Bloat Problem in A2A (Agent-Agent Communication)

The Agent-to-Agent (A2A) protocol defines structured message exchanges among multiple AI agents to support coordination, collaboration, and task delegation. It provides a semantic framework through which agents can advertise their capabilities, exchange intermediate reasoning results, and invoke each other's functions in a standardized manner.

A2A communication typically uses JSON-based message structures to ensure interoperability across heterogeneous agents. However, as the number and complexity of agents increase, message size and redundancy also grow. Agents often repeatedly exchange static or overlapping information (such as capability descriptions, metadata, or schema definitions) across multiple rounds of interaction. This repetition leads to a phenomenon known as token bloat, where verbose or redundant message components consume excessive tokens in the LLM's context window.

Token bloat degrades reasoning efficiency, increases latency, and elevates operational cost, particularly in large-scale or multi-agent deployments. These challenges highlight the need for a unified and optimized data layer that reduces redundancy while preserving interoperability and semantic integrity.

1.2. Token Bloat Problem in MCP (Agent-Tool Communication)

The Model Context Protocol (MCP) defines a standardized interface for communication between Large Language Models (LLMs) and external tools or data sources. It allows agents to describe, list, and invoke tools in a structured and interoperable manner. MCP follows a client-server architecture consisting of a client (which connects to the MCP server), a server (which hosts tool definitions and logic), and a host (which coordinates the overall interaction). Conceptually, MCP consists of two layers:

1. the data layer, which defines JSON-RPC-based primitives such as tools, resources, prompts, and notifications; and

2. the transport layer, which handles connection establishment, message framing, and authorization.

As the number of MCP servers/tools increases, token bloat problem also emerges. Each tool's detailed schema, including optional fields and human-readable descriptions, is transmitted to the model during reasoning. Overlapping schemas, unfiltered tool lists, and verbose outputs also contribute to redundant token usage. Though protocol correctness is unaffected, efficiency, latency, and cost are negatively impacted. These limitations motivate the design of a more efficient data layer.

1.3. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Problem Statement

Agentic communication introduces unique efficiency challenges because all exchanged messages are recursively processed by Large Language Models (LLMs). Every transmitted token consumes part of the model's limited context window, making redundancy and verbosity directly detrimental to reasoning efficiency and operational cost.

Across current agent protocols - such as the [MCP] and [A2A] - four common inefficiencies are observed in the data-layer design: repetitive schemas, excessive optional content, verbose responses, and inefficient capability selection. Together, these issues cause token and context bloat, limiting scalability and responsiveness in multi-agent systems.

2.1. Repetitive Schema Across Tools

When the LLM requests a list of tools from the server, the server returns a list of tools that includes the tool schema for all tools. However, tools hosted on the same MCP server often share similar or identical schema components, such as parameters or output schemas. These recurring elements not only introduce unnecessary repetition but also waste tokens when provided to LLM. We have conducted an experiment to analyze the duplicate content in the schemas of 60 tools within the official GitHub MCP server. According to our results, the repetition rate of some contents within the 60 tool schemas, such as

```
{
  "repo": {
        "description": "Repository name",
        "type": "string"
      }
}
```

Figure 1: Repetitive Schema Example

reached 9.84%, indicating that it is necessary to improve the tool schemas to reduce such repetitive fields.

2.2. Excessive Optional Content

The MCP tool schema currently includes a variety of optional fields, such as output schemas and metadata. In many cases, these fields are not essential for successful tool invocation. For instance, the output schema is primarily intended to help the LLM interpret and structure the server's response. However, when the server's output is already concise and semantically clear, the value of providing an additional output schema becomes marginal. From the server's standpoint, these fields are optional; yet, once transmitted, they are fully processed and tokenized by the LLM, contributing directly to prompt length and token overhead. At present, the client has no mechanism to indicate whether such optional content should be included, resulting in unnecessary data exchange and reduced token efficiency.

2.3. Overly Long Server Responses

Under the current MCP architecture, the server is responsible for generating outputs that conform to its declared output schema. However, in many scenarios, this design results in overly verbose responses that exceed the LLM's actual informational needs. For example, when a weather-reporting tool provides detailed structured data—including temperature, humidity, wind speed, and visibility—an LLM may only require a subset of this information, such as temperature and humidity. Since the server lacks awareness of the LLM's contextual requirements, it always returns the complete output defined by the schema. Consequently, the LLM must parse and tokenize all fields, even those irrelevant to the current prompt. This behavior introduces unnecessary token consumption and increases computational cost, particularly in use cases where concise responses are sufficient.

2.4. Inefficient Tool Selection

In practice, a service provider typically exposes a single MCP server that aggregates multiple tools under one endpoint. These tools may span heterogeneous functional domains (e.g., translation, data retrieval, summarization), all of which are simultaneously discoverable by the MCP client. While this design simplifies server-side deployment and management, it introduces inefficiency on the client side. As the number of available tools grows, agents must evaluate and compare an increasingly large set of tool descriptions to determine the most relevant one. This process not only increases computational overhead but also leads to selection ambiguity, especially when multiple tools exhibit overlapping functionalities or verbose metadata. Empirical observations indicate that excessively detailed tool descriptions, while intended to enhance interpretability, can paradoxically increase token consumption during parsing and prompt formation. Moreover, the expanded search space degrades the accuracy and efficiency of tool selection, particularly in real-time or resource-constrained scenarios.

3. Solution Overview

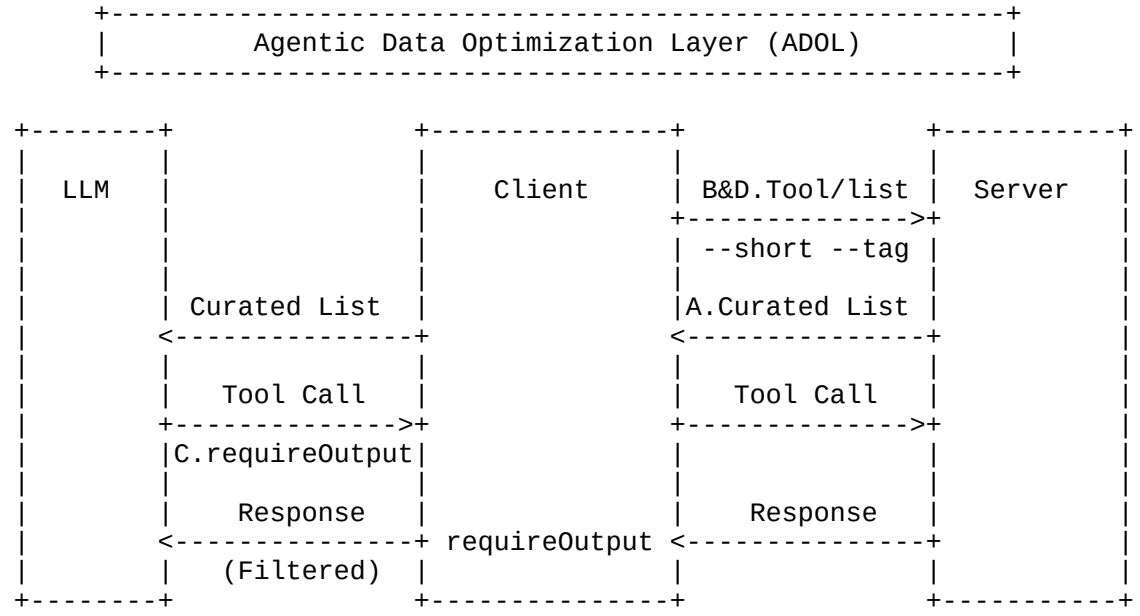

Figure 2: Typical workflow of ADOL

To systematically address the inefficiencies identified above, this draft introduces the Agentic Data Optimization Layer (ADOL) -- a unified and backward-compatible enhancement to existing agentic data layers. Rather than focusing on a single protocol, ADOL provides a generalized framework that improves the efficiency of data exchange across diverse agent communication settings, including both Model Context Protocol (MCP) and Agent-to-Agent (A2A) workflows.

ADOL defines a set of lightweight mechanisms that preserve interoperability while significantly reducing redundant token usage in LLM-driven interactions. It optimizes data compactness and adaptiveness without altering existing transport or semantic layers. As displayed in Figure 2, ADOL enhances the agentic data layer along four complementary dimensions:

A. Schema Deduplication: eliminate repetitive definitions through JSON references;

B. Adaptive Optional Inclusion: allow clients to request concise or full schemas;

C. Controllable Response Verbosity: enable models to specify the required output scope;

D. Context-Aware Tool Selection: limit tool exposure to those relevant to the current task.

Together, these mechanisms establish a more efficient and extensible foundation for agentic communication, reducing token and context bloat while maintaining full compatibility with existing agent frameworks.

3.1. Schema Deduplication Using JSON References

To eliminate redundant schema definitions, MCP tool schemas may enable the JSON ‘$ref‘ mechanism, which was not supported in the original MCP architecture and thus leads to a significant amount of duplicate content. This JSON ‘$ref‘ mechanism supports both internal and external references, as can be seen in [JSON].

```
{
  "definitions": {
    "user": { "type": "object" }
  },
  "properties": {
    "data": { "$ref": "#/definitions/user" }
  }
}
```

Figure 3: Example: Internal Reference

The internal reference case enables schema reuse within the same file, which applies to situations where there is multiple duplicate content within the same Json schema.

```
{
  "$ref": "schema/common/user.json"
}
```

Figure 4: Example: External Reference

The external reference allows a tool schema to reference a shared template stored either locally or remotely. The MCP host or client may cache these schemas to avoid repeated token consumption. This mechanism is more suitable for situations where there is overlapping content among multiple tool schemas. We have conducted extensive experiments where the tool schema in the server are modified by using $ref references. The results show that the LLM can still accurately recognize and invoke these tools, introducing no backward incompatibility.

3.2. Adaptive Optional Field Inclusion

When responding to ‘tool/list‘ requests, servers SHOULD determine whether to include optional schema content based on the client request. Specifically, an additional field ‘optional‘ is introduced in the server’s tool configuration file, which contains schema content that is not mandatory. If the client requests a “short” list (for example, via ‘tool/list --short‘), the server SHOULD omit unnecessary optional contents included in the ‘optional‘ field and return a concise tool schema. This conditional inclusion mechanism prevents unnecessary token usage.

3.3. Controllable Response Verbosity

Client SHOULD be capable of adjusting response verbosity based on the LLM’s request. To address this, ADOL introduces a new ‘requireOutput‘ field in the tool schema.

```
{
  "requireOutput": {
    "description": "A list of output field names that the LLM expects to be returned f
rom the tool's complete output.",
    "type": "array",
    "items": {
      "type": "string",
      "enum": ["toolOutcome_1", "toolOutcome_2", "toolOutcome_n"]
    },
    "uniqueItems": true
  }
}
```

Figure 5: RequireOutput Schema

When invoking a tool, the LLM first retrieves the tool list and corresponding schemas through the client. Each tool schema may include a 'requireOutput' field, which defines an optional list of output fields that can be selectively requested. This field enables the LLM to specify which parts of a tool's output are relevant to the current context. Based on this field, the LLM infers which elements of the tool's defined enumerated outcome are needed and passes the generated 'requireOutput' schema to the MCP client. Upon receiving the tool's execution result, the MCP client filters the output according to 'requireOutput' and returns the trimmed result to the LLM. For example, in a weather-related tool, the output schema may contain multiple fields such as current conditions, temperature, humidity, and visibility. The 'requireOutput' allows the client to specify which subset of these results should be included (e.g., '"requireOutput": ["temperature", "humidity"]'). After receiving the server's response, the client extracts and forwards only these fields to the LLM, filtering out unnecessary content while preserving semantic completeness. This design ensures that the LLM only receives the necessary data for the current context, thereby reducing redundant information and minimizing token consumption.

3.4. Tool Selection Mechanism

To alleviate the token overhead caused by large tool lists, an retrieval-based mechanism is introduced. Generally, The MCP client may include additional informative field in the 'tool/list' request to indicate its requirements for tool. The server then filters and returns only the tools that meet the need. We propose two typical implementation approaches:

1. Semantic-based retrieval: The client provides a natural language description of its requirement. The server computes an embedding for the requirement and compares it with precomputed embeddings of all available tools. Then, the server ranks the tools based on similarity and returns either the top-K most relevant tools or those whose similarity score exceeds a predefined threshold.

2. Tag-based retrieval: The client provides one or more predefined tool tags. Each tool is assigned at least one tag. The server returns the tools whose tags match the ones requested by the client.

These two approaches can accommodate servers with different capabilities. Approach 1 requires a more sophisticated server implementation, but it is expected to provide better retrieval performance and higher flexibility. In contrast, Approach 2 is simpler to implement, though it introduces some usage limitations.

By tool selection mechanism, the range of tools selected by the LLM is narrowed down, thereby reducing the token overhead of the LLM and improving the accuracy of tool selection.

4. Specification

The proposed extensions can be summarized as follows:

Mechanism	Description	Impact
JSON $ref support	Deduplicate schema content	Reduce token usage
Optional field control	Client-specified schema inclusion	Reduce token usage
Response Verbosity control	Adaptive detail level in output	Reduce token usage
Retrieval-based tool selection	Vectorized tool selection	Improve accuracy

Table 1: Mechanism Summary

Backward compatibility is maintained, as none of the proposed changes modify existing MCP schema semantics.

5. IANA Considerations

This memo includes no request to IANA.

6. Security Considerations

Embedding-based retrieval introduces potential information exposure risks if tool embeddings or client embeddings are transmitted in plaintext. Implementations SHOULD ensure encryption and access control when exchanging embedding vectors. Schema caching and external references should only reference trusted sources to avoid code injection or schema poisoning.

In addition to this, ADOL does not introduce new security vulnerabilities beyond those already present in existing agent communication protocols such as MCP or A2A. However, implementers should ensure that any schema references or external resources retrieved via $ref are obtained from trusted sources to prevent schema injection or data tampering.

7. References

7.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

7.2. Informative References

[A2A] Google, "Agent2Agent(A2A) Protocol", 2025, <https://a2a-protocol.org/latest>.

[MCP] Anthropic, "Model Context Protocol (MCP)", 2025, <https://modelcontextprotocol.io/docs/getting-started/intro>.

[JSON] Wright, A., Andrews, H., Hutton, B., and G. Dennis, "JSON Schema: A Media Type for Describing JSON Documents", 2022.

Acknowledgements

The author thanks the MCP and A2A open-source community for the ongoing discussion and the SEP authors for foundational design insights. The SEP link we proposed in the official MCP GitHub community is: https://github.com/modelcontextprotocol/modelcontextprotocol/issues/1576.

Authors' Addresses

Zeze Chang
Huawei
No. 3, Shangdi Information Road, Haidian District
Beijing
100085
China
Email: changzeze@huawei.com

Jinyang Li
Huawei
No. 3, Shangdi Information Road, Haidian District
Beijing
100085
China
Email: lijinyang9@huawei.com

Zhen Cao
Huawei
No. 3, Shangdi Information Road, Haidian District
Beijing
100085
China
Email: zhen.cao@huawei.com