A Large-Scale Dataset of MCP Implementations on GitHub

Benny Toeppe
Oakland University
Michigan, USA
btoeppe@oakland.edu

Amine Barrak
Oakland University
Michigan, USA
aminebarrak@oakland.edu

Emna Ksontini
University of North Carolina
Wilmington
North Carolina, USA
ksontinie@uncw.edu

Abstract

The rapid emergence of the Model Context Protocol (MCP) has introduced a new standard for connecting large language models to external tools and services. Despite its rapid adoption in open-source development, systematic understanding of how MCP is implemented, structured, and maintained remains limited. This study presents the first large-scale, evidence-based dataset of real-world MCP implementation collected directly from GitHub. Using a hybrid pipeline that integrates the GitHub REST and GraphQL APIs with custom Python verification scripts, 3,238 candidate repositories were discovered, filtered, and validated through multi-stage evidence checks. Each verified project was classified by operational role (e.g., client, server, gateway) and exported in a reproducible JSONL schema. A manual review of a representative subset confirmed an overall precision of 83% at a 95% confidence level, and additionally revealed a set of repositories functioning primarily as educational samples, tutorials, or demonstration templates. A targeted exclusion rule was then applied to remove these non-operational repositories, resulting in a final dataset of 2,297 validated MCP projects. The analysis shows that Python and TypeScript dominate MCP development, with hybrid architectures emerging as the most common design pattern. By emphasizing transparent verification strategies, structured evidence tagging, and reproducible data organization, this work establishes a foundational benchmark for studying real-world MCP ecosystems and supports future research on integration, connectivity, and compatibility across the broader developer community.

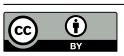
CCS Concepts

• **Software and its engineering** → **Software repositories and source code management**; • **Information systems** → *Data mining*; • **Computing methodologies** → *Machine learning*.

Keywords

Model Context Protocol (MCP), dataset, GitHub, open source software, repository mining, GraphQL, gateways

ACM Reference Format:
Benny Toeppe, Amine Barrak, and Emna Ksontini. 2026. A Large-Scale Dataset of MCP Implementations on GitHub. In *23rd International Conference on Mining Software Repositories (MSR '26), April 13–14, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3793302.3793311



MSR '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2474-9/2026/04
https://doi.org/10.1145/3793302.3793311

1 Introduction

Modern language model systems are moving from pure text generation to agents that use tools and live data, enabling them to plan tasks, call external services, and work with information that changes over time rather than relying only on training data [2, 7]. Early attempts to connect LLMs with external tools relied on bespoke, ad hoc solutions such as proprietary function-calling mechanisms and custom API integrations [8, 9]. These proved that models could call external functions but created significant and unsustainable friction for developers, since every new tool or data source required custom integration code that tightly coupled the tool to a specific model or host application [6]. A more scalable approach is to let agents communicate with tools and data sources through a shared protocol. The Model Context Protocol is the most visible effort in this direction. It defines how an application with an embedded model discovers capabilities and invokes them through a simple client–server design, allowing any compliant client to discover and call any compliant server through a consistent interface [3].

However, we still lack a clear, code grounded picture of how this protocol is implemented at scale. Public documentation and vendor posts explain the protocol and its goals, but they do not show which concrete design choices developers make in repositories, how those implementations evolve over time, or what evidence links a listed project to a working client or server. Early empirical work confirms why this gap matters. Hasan et al. [4] analyze 1,899 servers and report protocol specific vulnerabilities and maintainability concerns using static analysis and an MCP focused scanner, which underscores the need for datasets that let researchers connect ecosystem level claims to the actual code that developers publish. To date, the main attempts to map the ecosystem at scale rely on registry or marketplace listings rather than on repository centric verification. Lin et al. [5] introduce MCPCorpus by starting from MCP.so and then enriching entries with GitHub level signals such as stars, forks, contributors, and last commit time. This market centered efforts are valuable for ecosystem measurement, yet they do not provide a repository first and verifiable ground truth of MCP implementations that is built from code evidence in the repositories themselves.

In this paper, we construct a large scale repository centered dataset of MCP implementations collected directly from GitHub. Our pipeline discovers candidate repositories with the GitHub REST and GraphQL APIs, then applies multi stage, evidence based verification to confirm that each project is a working server, client, or gateway. The verification uses tangible code level signals, including dependency fingerprints from package manifests and protocol specific file layouts, and it records these signals as evidence for each entry. A manual review over a representative sample confirms the precision of the pipeline, which supports reproducibility and transparent curation. The final dataset contains 2,297 validated

projects out of 3,238 discovered candidates, organized in a reproducible JSONL schema. Beyond point in time metadata, we also collect change history to support evolution studies, including commit records and pull requests that modify MCP related files and configuration paths.

The complete dataset, including implementation evidence, classification outputs, and code, is publicly accessible at [1].

2 Dataset Construction

Figure 1 summarizes our dataset construction pipeline. We began with 3,238 public repositories referencing the Model Context Protocol (MCP). After removing stale and documentation-only repositories, 3,058 remained. Our verification pipeline confirmed MCP-related code artifacts in 2,387 repositories. A final refinement step based on manual validation removed 90 non-operational projects, resulting in 2,297 verified MCP implementations.

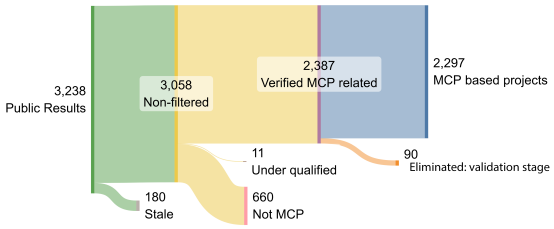


Figure 1: Flow of dataset construction from initial GitHub results to the final verified MCP implementations.

2.1 Initial Repository Discovery

We collected candidate repositories directly from GitHub using the REST Search API combined with GraphQL metadata retrieval. To ensure relevance to the emergence of MCP-driven agentic workflows, we limited the dataset to repositories created or updated between **January 2024** and **October 2025**.

We designed five search queries to capture both broad references to MCP and more specific implementation contexts. Two queries targeted explicit references to the protocol, while three focused on contexts where MCP usage is strongly associated with implementation, such as Claude Desktop integrations and server or client role naming. Table 1 summarizes the total results returned from GitHub and the number retained after filtering for duplicates, false positives, and archived projects.

Table 1: GitHub search queries and resulting repository

Query	Total repositories (response)	Total repositories (Kept)
`"MCP" in:name,readme,description`	3,148	2,379
`"mcp server" in:name,readme,description`	2,410	1,999
`"Model Context Protocol" in:name,readme,description`	1,815	1,521
`"Claude Desktop" MCP in:name,readme,description`	1,093	979
`"mcp client" in:name,readme,description`	800	743

After merging overlapping results across queries and removing duplicates, we obtained 3,238 unique repositories.

2.2 Filtering for Active Maintenance

To ensure that the dataset reflects active MCP development rather than abandoned prototypes or documentation, we applied the following filters:

- Repositories with no commits within the last nine months were removed.
- Repositories consisting solely of documentation, tutorials, or announcement stubs were removed.
- Forks without modification beyond the upstream version were excluded.

This resulted in a refined candidate set of 3,058 repositories.

2.3 Multi-Stage Verification Pipeline

We applied a verification pipeline designed to detect operational MCP implementations. Each repository was examined across three evidence layers:

2.3.1 Manifest and Dependency Evidence. We searched ecosystem-specific manifest and lock files to identify MCP-related dependencies, including:

- JavaScript: `package.json`, `package-lock.json`, `yarn.lock`
- Python: `pyproject.toml`, `Pipfile.lock`, `poetry.lock`
- Rust, Go, PHP: `Cargo.toml`, `go.mod`, `composer.json`

Common dependency indicators included: `mcp/sdk`, `fastmcp`, `mcp`, and `mcp-client`.

2.3.2 Directory Structure and Entry Points. Because manifests alone do not ensure operational implementation, we analyzed:

- Directory patterns such as `/src/mcp/`, /cmd/mcp/, and /bin/mcp-server.js
- Executable entry points configured via script definitions (e.g., `"mcp:run"`, `mcp-server`, `[project.scripts]`)

Repositories exhibiting MCP entry points were marked as executable MCP systems.

2.3.3 Integration and Environment Evidence. We inspected deployment environments via:

- CI/CD workflows (e.g., `.github/workflows`)
- Containerization files (e.g., `Dockerfile`, `docker-compose`)
- Claude Desktop configuration files (e.g., `config.json`)

This step identified 660 repositories with no operational MCP artifacts, which were removed, along with 11 incomplete implementation scaffolds. This resulted in 2,387 verified MCP repositories.

2.4 Manual Validation and Rule Refinement

To evaluate the automated verification stage, two authors independently reviewed the 170-repository subset, which was selected at the 95% confidence level with an estimated margin of error of ±7–8%. Each author examined whether the project implemented MCP by inspecting configuration files, dependency declarations, and entry-point code. Disagreements were resolved through discussion to reach a final consensus, reducing individual bias. This validation revealed several recurring sources of false positives:

- Repositories using `"MCP"` to refer to unrelated systems (e.g., Minecraft Coder Pack).

- Reference-only mentions of MCP without executable server or client code.
- Template repositories with placeholder code and no functional implementation.

The manual assessment between the two authors found that 83.0% of the reviewed repositories represented genuine MCP implementations, while 12.0% were false positives and 5.0% were false negatives. Most errors stemmed from language-specific file conventions or incomplete dependency manifests. Extrapolating these proportions to the initial set of 2,387 automatically verified repositories yields a Wilson confidence interval of [79.0%, 87.0%] at the 95% confidence level. We refined the filtering rules based on these findings, removing 90 additional false matches and producing a final dataset of 2,297 verified MCP implementations.

3 Dataset Contents and Characteristics

This section outlines the 2,297 verified MCP repositories, the operational roles they implement, how roles interact, and how role assignments and metadata were derived.

3.1 Role-Based MCP Execution Model

The Model Context Protocol defines how agents invoke external tools through a structured request-response workflow. In practice, implementations of MCP adopt one of three primary operational roles that correspond to different points in the execution chain. Figure 2 illustrates the interaction pattern among these roles.

- **Client**: initiates connections to MCP servers and consumes their tools, resources, or prompts. Typical examples include user-facing interfaces such as Claude Desktop integrations.
- **Server**: Exposes MCP-compatible tools, resources, or prompts that Clients can invoke. Servers implement the logic for computation, retrieval, transformation, or control when handling client requests.
- **Gateway**: Mediates MCP traffic between components or external ecosystems. Gateways translate, route, or relay requests, often bridging MCP with APIs.

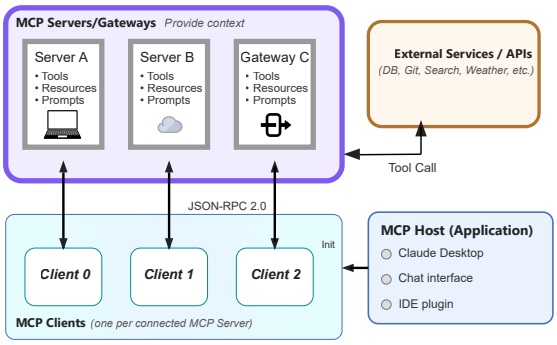


Figure 2: MCP architecture showing the host, clients, and servers/gateways exposing tools and resources.

3.2 Role Classification Rules

Each verified repository was assigned a primary operational role based on source-level evidence. The role classification used a rule-based scoring system combining manifest signals, entry-point patterns, and semantic cues from metadata and documentation, resulting in one of three roles: **Client**, **Server**, or **Gateway**.

Rule-Based Scoring System. Three independent score variables were computed for each repository: S_c (client score), S_s (server score), and S_g (gateway score). Signals were drawn from repository metadata, directory structure, source paths, dependency manifests, and commit-level evidence.

- **Client signals**: Use of MCP client libraries and explicit tool invocation (e.g., `connect()`, `invoke()`, `use_tool()`, paths such as `@mcp/sdk/client`), including editor plugin contexts (e.g., `vscode`, `cursor`, `claude_desktop`).
- **Server signals**: Registration and exposure of MCP tools or capabilities, indicated by directories like `/tools/` or /resources/, manifest references to `@mcp/sdk` or `fastmcp`, and server entrypoints such as `server.py`, `server.ts`, or mcp-server binaries.
- **Gateway signals**: Logic forwarding MCP requests across systems, indicated by terms such as `gateway`, `router`, `relay`, or `proxy`, and bridges to external APIs or services. Observed patterns fall into six subtypes: transport relay, language bridge, authentication policy, orchestration, sandbox isolation, and discovery/registry.

Each match contributed to a numeric score based on indicator strength (strong = 2 points, weak = 1 point), where strong signals derived from source and manifest files and weak signals from README/metadata text.

Decision Logic. The final role was determined based on the comparative magnitude of the three scores:

(1) If $S_c \geq 3$ and $S_s \leq 1$, classify as **Client**.
(2) If $S_s \geq 3$ and $S_c \leq 1$, classify as **Server**.
(3) If $S_g \geq 4$ and both S_c and $S_s < 2$, classify as **Gateway**.

A **client override rule** was applied to prevent misclassification of editor extensions and IDE integrations that contain incidental routing logic. If gateway-scored repositories contained editor signals (e.g., `vscode`, `cursor`, `claude_desktop`), the final classification was reassigned to **Client**.

Manual Disambiguation. Repositories where the scoring system did not yield a clear dominant role (e.g., when $|S_c - S_s| \leq 1$ or when all three scores were below threshold) were marked as *low confidence*. These repositories were reviewed manually to determine whether they primarily issued tool requests, exposed tools, or mediated between systems. If a dominant operational role could not be identified after inspection, the repository was labeled as **Unclassified**.

Applying these rules resulted in 1,962 Servers, 1,462 Clients, and 80 Gateways. Some repositories exhibited both client and server behavior and were assigned to both roles, while 36 could not be assigned a dominant role and were marked as **Unclassified**.

Among the 80 Gateways, orchestration gateways were the most common (n=47), followed by language bridges (n=22). Transport relays (n=6) and authentication policy gateways (n=5) appeared less

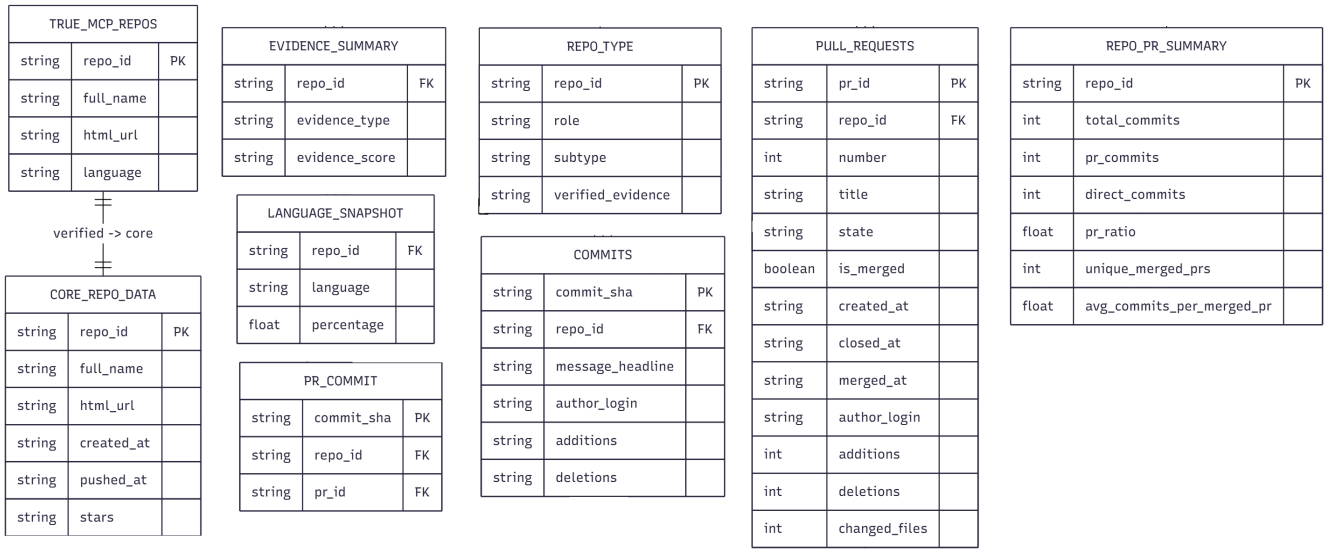


Figure 3: Dataset Schema and Entity Relationships for the MCP Repository Dataset

frequently, with sandboxing (n=2) and discovery/registry gateways (n=1) being rare.

The 36 Unclassified repositories were mostly lightweight or incomplete projects, including agent adapters (n=13), chat UI integrations (n=5), documentation or specification scaffolds (n=4), SDK utilities and CLI tooling (n=6), small demo or benchmark artifacts (n=4), and unstructured repositories (n=4).

3.3 Language Characteristics by Role

We examined the primary implementation language of each repository after role classification. Table 2 presents the top three languages for each role, ranked by number of repositories.

Table 2: Top 3 Languages per MCP Role

Rank	Server	Client	Gateway
1	Python (785)	Python (530)	TypeScript (25)
2	TypeScript (597)	TypeScript (492)	Python (22)
3	JavaScript (177)	JavaScript (137)	Go (17)

Python and TypeScript dominate both server and client implementations. JavaScript appears in smaller client- and server-side projects and examples. Gateways show a distinct profile, with Go appearing as a third major language.

3.4 Dataset Schema and Repository Metadata

Figure 3 shows the schema used to store the verified MCP repositories and their metadata. The dataset centers on a core repository index, with auxiliary tables for language composition, role classification, evidence signals, and pull request activity.

Repository Core and Verification Evidence. The dataset centers on the `TRUE_MCP_REPOS` table, which lists the 2,297 verified repositories. Each repository is linked to:

- `CORE_REPO_DATA`, containing metadata such as creation date, last push timestamp, and star count.
- `EVIDENCE_SUMMARY`, which records the types of signals that contributed to verification (manifest, entry points, directory patterns, etc.) along with evidence strength scores.
- `REPO_TYPE`, which defines the repository's assigned role (Client, Server, Gateway), subtype where applicable, and supporting evidence.

Commit-Level Activity. Development history is stored in the `COMMITS` table, which records commit messages, author identity, and code change size. Across all verified repositories, we observe:

- Median commits per repository: **70**
- Median commit message length: **38 characters**
- Median additions per commit: **19 lines**
- Median deletions per commit: **4 lines**
- Median changed files per commit: **2 files**
- Median number of unique contributors per repository: **4**
- Median repository stars: **156**
- Median forks per repository: **27**
- Median open issues per repository: **3**
- Median number of published releases: **1**

These patterns suggest iterative, small-step development styles rather than large batch updates.

Pull Request Structure and Collaboration. Pull request metadata is stored in the `PULL_REQUESTS` table, with aggregate summaries in `REPO_PR_SUMMARY`. We track counts of merged PRs, review patterns, and author involvement. At the dataset level, we observe:

- Most MCP servers and clients are maintained by **small teams** (2–5 active contributors).
- Gateway repositories show **higher PR activity**, with an average of 49.6 pull requests per repository, compared to 33.7 for clients and 29.9 for servers.

4 Conclusion

We constructed a dataset of 2,297 verified MCP repositories using a multi-stage discovery and verification process based on source-level evidence. The dataset records role assignments, evidence indicators, language composition, and development activity in a structured format. Python and TypeScript are the most common languages in both client and server implementations, while gateways additionally use Go. Several repositories include both client and server functionality within the same codebase. The dataset is intended to support reproducible analysis of MCP implementation practices, with future work focusing on updating the dataset over time and refining role and subtype classifications.

References

[1] anonymous. 2025. *Verified MCP Implementations Dataset.* doi:10.5281/zenodo.17573071

[2] Amine Barrak. 2025. Traceability and Accountability in Role-Specialized Multi-Agent LLM Pipelines. In *Proceedings of the MAS-GAIN Workshop at the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE.

[3] Mohamed Amine Ferrag, Norbert Tihanyi, and Merouane Debbah. 2025. From llm reasoning to autonomous ai agents: A comprehensive review. *arXiv preprint arXiv:2504.19678* (2025).

[4] Mohammed Mehedi Hasan, Hao Li, Emad Fallahzadeh, Gopi Krishnan Rajbahadur, Bram Adams, and Ahmed E Hassan. 2025. Model context protocol (mcp) at first glance: Studying the security and maintainability of mcp servers. *arXiv preprint arXiv:2506.13538* (2025).

[5] Zhiwei Lin, Bonan Ruan, Jiahao Liu, and Weibo Zhao. 2025. A Large-Scale Evolvable Dataset for Model Context Protocol Ecosystem and Security Analysis. *arXiv preprint arXiv:2506.23474* (2025).

[6] Meriem Mastouri, Emna Ksontini, and Wael Kessentini. 2025. Making rest apis agent-ready: From openapi to mcp servers for tool-augmented llms. *arXiv preprint arXiv:2507.16044* (2025).

[7] Salvatore Raieli and Gabriele Iuculano. 2025. *Building AI Agents with LLMs, RAG, and Knowledge Graphs: A practical guide to autonomous and modern AI agents.* Packt Publishing Ltd.

[8] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems* 36 (2023), 68539–68551.

[9] Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *arXiv preprint arXiv:2306.05301* (2023).