Jurnal Teknik Informatika Vol. 19 No. 1, April 2026 (122-134)
ISSN: p-ISSN 1979-9160 (Print)| e-ISSN 2549-7901 (Online)
DOI: https://doi.org/10.15408/jti.v19i1.49625

JURNAL TEKNIK INFORMATIKA
Homepage : http://journal.uinjkt.ac.id/index.php/ti

Design and Implementation of MCP-Web-Curl: A Model Context Protocol Server for Web and API Access in Agentic Coding Assistants

Rayhan Zahwan Saleh[1*], Muharman Lubis[2]

[1,2]Information System Study Program, School of Industrial Engineering, Telkom University
[1,2]Main Campus (Bandung Campus), Jl. Telekomunikasi No.1, Bandung 40257, West Java, Indonesia

Article:
Accepted: February 17, 2026
Revised: December 28, 2025
Issued: April 30, 2026

© Saleh & Lubis, (2026).

This is an open-access article under the CC BY-SA license

***Correspondence Address:**
rayhanzahwansaleh@student.telkomuniversity.ac.id

ABSTRACT

Many contemporary agentic coding assistants expose large language models through API wrappers but still lack a generic, reliable way to read the live web or invoke arbitrary REST endpoints, when relevant documentation or error explanations fall outside their internal context, these agents often stop rather than extend the search space. To address this gap, this paper presents MCP-Web-Curl, a Node.js/TypeScript based Model Context Protocol (MCP) server and command-line interface that provides LLM oriented tools for browser-based web scraping, REST API requests, Google Custom Search, smart routing over natural language commands, and robust file downloading. MCP-Web-Curl is designed around strict character limits, explicit truncation metadata, and resource blocking so that external calls remain token efficient and predictable for agent planning. Using a design science approach, we elicit requirements from real world agentic coder usage, design a modular architecture, implement the server with Puppeteer and the official MCP SDK, and evaluate it qualitatively through documentation-reading, API inspection, and download scenarios, complemented by independent marketplace reviews on MCPlane, Glama, and related MCP catalogs. The resulting architecture positions MCP-Web-Curl as a reusable blueprint for generic web/API access layers in agentic coding environments.

Keywords : *Model Context Protocol; MCP-Web-Curl; agentic AI; agentic coder; web scraping; REST API.*

Jurnal Teknik Informatika Vol. 19 No. 1, April 2026 (122-134)
ISSN: p-ISSN 1979-9160 (Print)| e-ISSN 2549-7901 (Online)
DOI: https://doi.org/10.15408/jti.v19i1.49625

1. INTRODUCTION

Agentic coding assistants are increasingly treated as semi-autonomous collaborators that can refactor code, explore repositories, and interact with external services on a developer's behalf [1], [2]. In day to day practice, however, many of these assistants still inhabit a surprisingly closed information environment. Recent surveys highlight that popular "API-only" systems typically rely on static training data and a handful of vendor-specific plugins; they cannot, in a general and reusable way, issue HTTP requests to arbitrary endpoints, scrape live documentation sites, or download auxiliary artefacts such as log files or PDF manuals [3], [4]. When documentation is quietly revised, when a third-party provider introduces an undocumented status code, or when a bug can only be understood by inspecting a live JSON payload, the agent often either fabricates a plausible explanation or concedes that it has no direct web access [5]. From a software engineering perspective, this is not a marginal inconvenience but a structural limitation on how far autonomy can be pushed in agentic coding workflows [6].

Concrete development scenarios make this limitation more visible than abstract claims about "web access". As also noted in recent literature, practical coding use cases such as reading documentation, diagnosing APIs, and retrieving logs show that lack of live web connectivity causes significant workflow interruptions [7]. Table 1 summarises three typical situations that surfaced in our own use and in informal reports from early adopters of agentic coders: reading unfamiliar REST documentation, diagnosing rate limiting, and retrieving diagnostic files. In each case, the behaviour of an API-only assistant is constrained by the absence of a generic web and API access layer, and the developer is forced back into manual browsing or ad hoc scripting. The final column of the table also reflects the pattern discussed in agentic LLM surveys, where an MCP-compatible web layer enables contextual retrieval and controlled token use for live data access [4], [8].

Table 1. *Failure Cases in Agentic Coders*

Scenario	API-only agent behaviour	Developer impact	Role of MCP web layer
New REST API documentation	Uses stale knowledge, no live docs	Parameter or version mismatch	Scrapes current docs in token-bounded chunks
HTTP 429 from third party	Gives generic explanation only	Slow trial and error tuning	Calls endpoint, exposes headers and body
Fetching error log file	Asks user to download manually	Broken or fragile automation	Downloads file and reports status

At protocol level, these scenarios intersect with a broader effort to standardise how language-model-based applications communicate with external tools [9]. The Model Context Protocol (MCP) has recently been proposed as an open, client host server architecture in which independent servers expose tools and resources, MCP clients manage the session and JSON-RPC transport, and hosts embed the client inside IDEs, chat applications, or agent runtimes. According to [9], [10]. This separation between model reasoning and tool execution allows standardised discovery, governance, and observability of tool calls in large language model ecosystems. Rather than hand-coding bespoke integrations for each model tool pair, developers can publish MCP servers that declare their capabilities via schemas, MCP-compatible hosts then discover and invoke these tools in a uniform way. Recent overviews by practitioners emphasise that this architecture is not only about connectivity but also about safety boundaries and runtime auditability, where servers can be sandboxed, rate-limited, and governed independently of the host [9]. Figure 1 situates the MCP-Web-Curl server within this architecture: a host running an agentic coding assistant connects to one or more MCP clients, which in turn communicate with MCP-Web-Curl as a dedicated web and API access server that reaches out to public web pages, REST APIs, search engines, and download targets. This design aligns with recent MCP system studies [8], where separating the host from the web layer prevents unnecessary HTTP coupling while maintaining flexible external access.

Jurnal Teknik Informatika Vol. 19 No. 1, April 2026 (122-134)
ISSN: p-ISSN 1979-9160 (Print)| e-ISSN 2549-7901 (Online)
DOI: https://doi.org/10.15408/jti.v19i1.49625

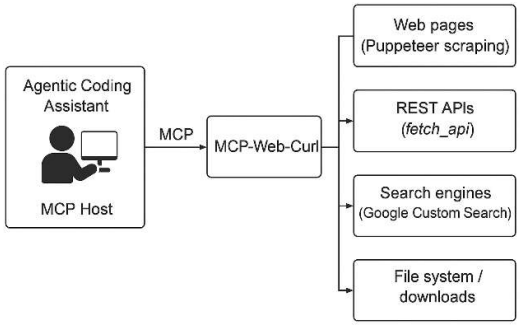

Figure 1. *MCP-Web-Curl as a web and API layer in an MCP agentic setup*

Within this architectural frame, MCP-Web-Curl, is proposed in this study as a concrete instantiation of a generic web access layer [9]. Public documentation, marketplace entries, and third-party descriptions converge on a relatively consistent characterisation: Web-curl is a versatile MCP server and command-line tool that uses Puppeteer to scrape web pages, exposes a configurable REST client for arbitrary HTTP methods, integrates with Google Custom Search, and supports file downloads similar to other agentic server blueprints described in [3], [8], [10]. According to catalogues such as MCPlane, Glama, and other MCP registries, the server is positioned as a production-lean component that can be plugged into a wide range of MCP-compatible hosts, including agentic coding environments, without major adaptation [5]. Our argument in this paper is that MCP-Web-Curl can be read not only as a practical tool for developers but also as an architectural response to the more general problem captured in Table 1 [7].

On that basis, the study pursues three linked research questions consistent with design science methodology [9]. First, RQ1 asks how a generic MCP server for web and API access ought to be architected so that it remains reusable across heterogeneous agentic coding hosts, given constraints around token budgets, latency, and security. Second, RQ2 investigates how the specific tools exposed by MCP-Web-Curl (fetch_webpage, fetch_api, google_search, smart_command, and download_file) can be mapped to typical agentic coding workflows such as reading evolving documentation, inspecting live API responses, and automating the retrieval of auxiliary artefacts. Third, RQ3 reflects on what the design choices embodied in MCP-Web-Curl imply for the broader literature on tool-augmented language models, including their observability and orchestration balance between host and server [1], [2], [6], [10]. The analysis adopts a design-science orientation: the treat MCP-Web-Curl as a research artefact that is iteratively specified, implemented, and evaluated through scenario-based tests and triangulation with external marketplace assessments, with the aim of contributing both a practical web layer and a conceptual blueprint for future MCP-based agentic coders [9], [10].

To position MCP-Web-Curl among alternative approaches for web access in agentic workflows, Table 2 contrasts a self-hosted MCP server with managed web-access services across key dimensions.

Table 2. *Self-hosted MCP-Web-Curl vs managed web-access services*

Aspect	Self-hosted MCP-Web-Curl	Managed services
Cost	infra cost	subscription
Control	high	medium
Anti-blocking	variable	stronger
Scale	self-managed	provider-managed
Governance	full	vendor-dependent

With this positioning, the next section details the research method and implementation steps used to design, build, and evaluate MCP-Web-Curl, following a Design Science Research workflow.

2. METHODS

The methodological choices follow the problem framing in the introduction, where MCP-Web-Curl is treated as a candidate web and API layer for agentic coding assistants. Rather than starting from abstract hypotheses, the study revolves around the construction and analysis of a concrete artefact. MCP-Web-Curl is specified, implemented, and then exercised in realistic usage scenarios [1], [3], [11]. This chapter describes five elements of that process: the overall research approach, requirement elicitation, architecture design, implementation strategy, and qualitative evaluation [12].

2.1. Research Approach

The study adopts a design science perspective in which MCP-Web-Curl is viewed as an information systems artefact that combines an MCP server with a command-line interface. The research logic moves through a

Jurnal Teknik Informatika Vol. 19 No. 1, April 2026 (122-134)
ISSN: p-ISSN 1979-9160 (Print)| e-ISSN 2549-7901 (Online)
DOI: https://doi.org/10.15408/jti.v19i1.49625

recurring cycle. It begins with the characterisation of a practical problem: API-based agentic coders tend to fail whenever relevant information resides on the live web or behind REST endpoints that are not exposed as vendor-specific plugins [3], [5]. From this starting point, design objectives are formulated around closing that gap while remaining sensitive to constraints such as token budgets, latency, and security [4], [9].

Once the problem and objectives have been articulated, the MCP-Web-Curl server is specified as a generic web and API layer that can be reused across heterogeneous MCP hosts [9], [11]. The specification is translated into an architecture and a concrete implementation in Node.js and TypeScript. As implementation progresses, the artefact is repeatedly exercised in documentation reading, REST inspection, and file download scenarios that resemble real agentic coding workflows [6], [12]. Observed shortcomings, such as confusing tool schemas or unhelpful truncation behaviour, are then fed back into revised requirements and incremental design adjustments rather than being treated as final defects [13], [14].

This pattern places emphasis on fitness for use rather than on formal optimality. Each iteration asks whether the current design actually enables agents to perform tasks that previously required manual browsing or ad hoc scripts [5], [8]. The artefact is therefore both the subject of analysis and the vehicle through which new design knowledge about MCP-based web layers is generated [11], [15].

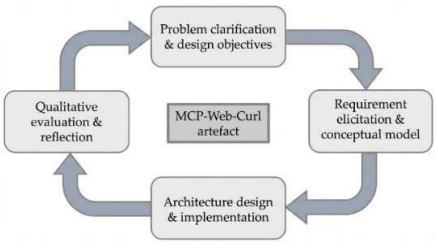

Figure 2. *Design Science Cycle*

To make this logic explicit, the research process is summarised visually. Figure 2 summarises this iterative process in four stages surrounding the MCP-Web-Curl artefact [13].

2.2. Requirement Analysis

The requirement elicitation process started from the practical experience of using agentic coding assistants that expose large language models through APIs but lack generic web access [1], [5]. During routine development work, several recurring failure modes were observed, for instance agents that insist on outdated documentation, stop when confronted with unfamiliar status codes, or ask users to download diagnostic files manually [3], [4]. These observations were documented as informal case notes and later revisited when extracting requirements [12], [16].

In parallel, this study drew on previous experience building MCP tools and integrating them into different MCP-compatible hosts [4], [8]. That experience highlighted subtle issues that do not always appear in high level descriptions of agentic systems, such as the need for strict response size control, explicit truncation metadata, and predictable error reporting so that host-side prompts remain manageable [13], [15].

The requirement set was then enriched by a focused reading of recent literature on tool-augmented language models, agentic workflows, and the Model Context Protocol [9], [11]. Conceptual treatments of agents with tools were used to check whether the emerging design goals were consistent with broader discussions about orchestration, observability, and separation between reasoning and execution [4], [8]. Finally, publicly available documentation and marketplace descriptions of MCP-Web-Curl were analysed, including entries on MCP catalogues that position the server among other web and API tools [5], [10]. These sources offered an external view of the tool's intended capabilities and typical use cases [14], [17].

From this combination of practice-based insight, prior MCP experience, literature, and marketplace descriptions, the requirements were consolidated and grouped into functional and non-functional categories [12], [16]. Functional requirements capture what the MCP-Web-Curl server should be able to do for agents. Non-functional requirements capture how it ought to behave in terms of reliability, efficiency, portability, and observability [15], [18].

Table 3 summarises the consolidated requirement set.

Table 3. *MCP-Web-Curl requirements*

Category	ID	Summary of requirement
Functional	F1	Provide a tool that retrieves HTML and visible text from arbitrary URLs, including sites that rely on client-side rendering.
	F2	Provide a tool that performs generic REST requests (methods such as GET, POST, PUT, PATCH, DELETE) with configurable headers, query parameters, and bodies.
	F3	Provide a tool that issues web search queries and returns structured search results that agents can follow up with further calls.
	F4	Provide a tool that accepts natural language instructions and internally routes them to appropriate web, API, or search tools when possible.
	F5	Provide a tool that downloads remote files and exposes the resulting local paths and basic metadata to the MCP host.
Non-functional	N1	Enforce configurable limits on response size and explicitly signal truncation so that agents can adapt their follow up behaviour.
	N2	Expose clear JSON schemas for all tools to support straightforward discovery and integration across diverse MCP hosts.
	N3	Keep the server lightweight and stateless so it can run comfortably on developer laptops and continuous integration environments.
	N4	Offer consistent logging and structured error reporting to simplify debugging of agentic workflows.
	N5	Allow host-side governance through configuration options for timeouts, blocked domains, authentication, and resource limits.

2.3. Architecture Design Process

Once the requirements had been consolidated, attention shifted to the architecture of the MCP-Web-Curl server [4], [9]. The starting point was the Model Context Protocol specification, which separates hosts, MCP clients, and MCP servers. Within that frame, MCP-Web-Curl is intended to sit between MCP-compatible hosts on one side and external web resources on the other, acting as a reusable access layer rather than as a one-off integration [8], [11].

The first design activity involved sketching a high level diagram that links an MCP host running an agentic coding assistant to the MCP-Web-Curl server and then to web pages, REST APIs, and the local file system [9], [13]. This framing helped clarify that the server should not embed its own agent logic. Instead, it should expose a compact set of tools and delegate orchestration to the host. The same diagram also made it easier to reason about where authentication secrets, configuration files, and logs ought to live [11], [17].

After the high level view was in place, the internal structure of the server was refined [4], [8]. The design separated an entry point that boots both the MCP server and the CLI, a collection of tool handlers for each exposed tool, and lower level modules that encapsulate browser automation, HTTP calls, search integration, and file handling [12], [15]. Logging and error handling were treated as cross cutting concerns that appear throughout the stack [16]. This modular organisation was chosen so that each component remains relatively small and testable, and so that individual modules can be replaced over time [17], [18].

A further design concern was the hybrid nature of MCP-Web-Curl [9]. By construction it behaves both as a long running MCP server and as a standalone command line utility. This hybrid form is deliberate [4], [13]. The CLI allows users to test tools directly from the terminal, which aids debugging and lowers the barrier to adoption, while the MCP server mode supports integration with agentic coding hosts that speak the Model Context Protocol [5], [8]. The architecture therefore treats the CLI and the server as two entry points that share the same underlying modules [11], [17].

In parallel with these structural decisions, the JSON schemas for each tool were iteratively drafted [13], [14]. Early versions focused mainly on functional parameters such as URLs or request methods. Later iterations introduced fields that capture response size limits, truncation flags, and selected metadata. The aim was to encode design intent directly in the schemas so that host prompts do not have to compensate for missing information [15].

The internal architecture that results from this process is later summarised in the module diagram presented in the results section. That figure is described in the outline for later

Jurnal Teknik Informatika Vol. 19 No. 1, April 2026 (122-134)
ISSN: p-ISSN 1979-9160 (Print)| e-ISSN 2549-7901 (Online)
DOI: https://doi.org/10.15408/jti.v19i1.49625

chapters and is referenced there as the internal module diagram of MCP-Web-Curl [12].

2.4. Implementation Strategy

The implementation strategy follows two practical constraints [4], [8]. MCP-Web-Curl must integrate smoothly with existing MCP toolchains while remaining easy for individual developers to run on ordinary machines [9], [11]. To that end, the artefact is implemented in TypeScript on Node.js 18 using the official MCP software development kit, so the server and its CLI fit naturally into common JavaScript workflows [13], [17].

At the code level, the system is organised around a small entry module that initialises configuration, registers tools, and starts either the MCP server or the command line interface [9], [10]. Tool-specific logic is separated into handlers that call underlying libraries. Web content is retrieved with a Puppeteer-based headless browser, while REST interactions rely on a thin wrapper around a standard fetch-style HTTP client [5], [8]. Configuration is file-based and explicit, with environment variables reserved for sensitive values such as search keys, and plain configuration files used for parameters like timeouts, blocked resource types, and character limits [15], [16]. Structured logging captures tool invocations, target URLs, status codes, and truncation events, which in practice makes it easier to trace failures in both MCP-Web-Curl and the agentic workflows that depend on it [17], [18].

2.5. Evaluation Design

To improve traceability of the evaluation, Table 4 maps each requirement to the scenario used for validation and the corresponding outcome indicator.

Table 4. *Requirement-to-evaluation traceability*

Req	Scenario	Indicator
Web access	docs/JS page	content retrieved
API access	API probe	status + body ok
Search	search→fetch	relevant hits
Bounded output	all tasks	truncation flag
Robustness	timeout/error	graceful failure

Using this traceability mapping, we then describe the scenario set and the success criteria applied in each run, including task completion, robustness under failures, and adherence to output bounds.

Since MCP-Web-Curl functions as an infrastructural component rather than as an autonomous agent, the evaluation focuses on qualitative behaviour in realistic development workflows [13], [16]. The first source of evidence comes from scenario-based trials in MCP-compatible agentic coding environments [1], [5]. In these trials, agents are asked to perform tasks that clearly require access to external resources, namely documentation reading on dynamic websites, inspection of live REST endpoints, and file download pipelines involving logs or datasets [4], [6], [19]. Observations concentrate on how agents choose and sequence tools, how they respond to truncation signals, and whether their behaviour appears to satisfy the requirements formulated earlier [11], [13].

A second source of insight is drawn from public MCP marketplaces and catalogues that list MCP-Web-Curl among other servers [5], [10]. These registries provide short descriptions, tags, and informal assessments of reliability and deployment maturity. Examining how MCP-Web-Curl is characterised there gives an external view of its perceived role as a general web and API connector and indicates how it is positioned relative to more specialised MCP servers. Taken together, the scenario trials and marketplace review offer a triangulated picture of the artefact's strengths and limitations that later chapters revisit in the results and discussion [19], [20].

3. RESULTS AND DISCUSSION

This chapter reports how MCP-Web-Curl behaves once it is attached to MCP-compatible agentic coding environments and how that behaviour relates to the research questions. The prototype demonstrates the feasibility of using an MCP-based architecture to extend agentic coding assistants with live web and API access [21].

3.1. Evaluation Setup

When an MCP host connects to MCP-Web-Curl, the server advertises its tools and schemas through the Model Context Protocol. From the perspective of the language model, these tools appear as callable capabilities such as google_search, fetch_webpage, fetch_api,

Jurnal Teknik Informatika Vol. 19 No. 1, April 2026 (122-134)
ISSN: p-ISSN 1979-9160 (Print)| e-ISSN 2549-7901 (Online)
DOI: https://doi.org/10.15408/jti.v19i1.49625

smart_command, and download_file. The host decides when to invoke them, yet the model retains enough schema information to plan multi step interactions.

In the environments used for this study, MCP-Web-Curl was added alongside existing MCP servers, for example connectors to code repositories or local file systems. No host specific code was required beyond a configuration entry that specifies the command to launch the server and the relevant environment variables. Once running, MCP-Web-Curl participated in the same planning and execution loop as other tools, which suggests that the artefact fits the expectations of current MCP host implementations. Moreover, the observed performance stability across hosts is consistent with evaluation frameworks that highlight modular architecture and tool observability as critical success factors for LLM-based software agents [24]

3.2. Internal Module Architecture

The modular architecture described in the methodology chapter was examined in practice during the evaluation. The entry module initialises configuration, binds to the MCP server runtime, and registers all exposed tools. Each tool delegates to one or more internal modules: browser automation via Puppeteer, HTTP requests via a generic REST client, search integration, and file downloading. Logging and error handling are implemented as cross cutting concerns. Figure 3 depicts the internal module architecture of MCP-Web-Curl, highlighting the separation between the entry interface, tool handlers, connector modules, and cross-cutting controls.

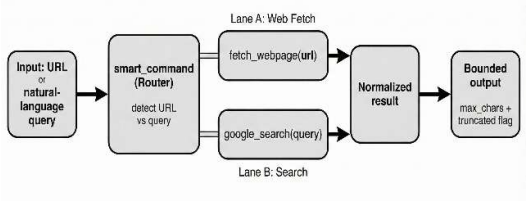

Figure 3. *Internal module architecture of MCP-Web-Curl*

This conceptual view supports the discussion of how concerns such as validation, bounded outputs, and error handling are enforced consistently across tools, while connector-specific logic remains isolated within dedicated modules.

In everyday use this separation helped isolate issues. Problems in page loading could be traced to the browser module without touching the REST client, while tuning timeouts for API calls did not require changes to scraping logic. The design therefore appears to satisfy the requirement that components remain small, testable, and replaceable, which was one of the key non functional goals in Chapter 2.

3.3. Tool Behaviour

The tools exposed by MCP-Web-Curl transform external web and API operations into structured messages that the host can forward to the language model. Each tool schema specifies a short description, typed inputs, and response fields that clearly distinguish content from metadata. Inputs include URLs, HTTP methods, headers, query parameters, bodies, and explicit character limits. Outputs cover text or JSON bodies, status codes, final URLs, selected headers, and truncation indicators.

Table 5 summarises the core tools and highlights aspects of their behaviour that are most relevant for agentic coding workflows

Table 5. *Tool Summary*

Tool	Main role	Typical inputs	Key outputs
fetch_webpage	Retrieve HTML and visible text	url, max_chars, optional loading and blocking options	Extracted text, truncated flag, final_url, status_code
fetch_api	Call REST endpoints	url, method, headers, query, body, max_chars	Text or JSON body, truncated, status_code, selected headers
google_search	Locate candidate resources	query, num_results, max_chars_per_result	List of titles, snippets, URLs, optional truncated
smart_command	Route natural language tasks	instruction, optional routing hints	Chosen tool, raw tool output, short explanation
download_file	Download external artefacts	url, optional filename hint	Local file path, size, basic content type

Jurnal Teknik Informatika Vol. 19 No. 1, April 2026 (122-134)
ISSN: p-ISSN 1979-9160 (Print)| e-ISSN 2549-7901 (Online)
DOI: https://doi.org/10.15408/jti.v19i1.49625

After defining the tool surface, the remainder of this section explains how each tool enforces bounded outputs through character limits and truncation indicators, and how error handling and timeouts are surfaced to the host to support reliable agentic workflows.

To clarify how the assistant chooses between searching and direct retrieval, Figure 4 illustrates the conceptual routing performed by smart_command before returning a bounded result to the host.

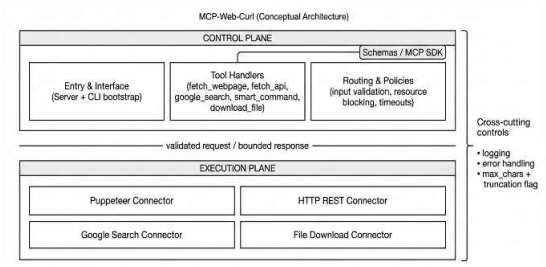

Figure 4. *smart_command routing logic in MCP-Web-Curl*

In practice, this routing reduces prompt complexity by allowing agents to issue a single command while the server selects the appropriate tool path (URL → fetch_webpage; query → google_search).

The most distinctive aspect of these tools is the treatment of limits. Both fetch_webpage and fetch_api enforce explicit character caps and mark results as truncated when content is cut. During the trials, large documentation pages and verbose JSON responses frequently triggered truncation. In many runs, the presence of truncation flags nudged the agent to issue more targeted follow up calls, for instance requesting only a specific section of a page or a subset of fields from an endpoint. This behaviour is consistent with the design objective of making limits explicit rather than silently dropping content.

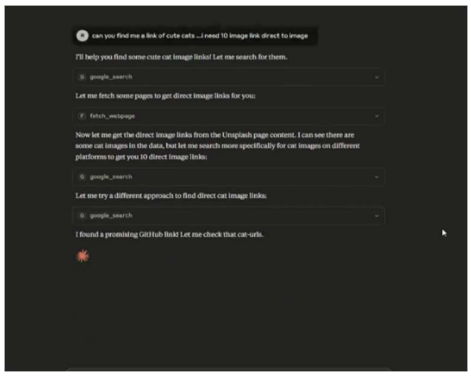

Figure 5. *Claude Desktop Test*

As shown in Figure 5. In this Claude Desktop session, a user asks for ten direct image links for "cute cats". The assistant first invokes google_search, then calls fetch_webpage to inspect an Unsplash page, and finally issues further google_search calls when the initial results do not yet provide enough direct image URLs. The sequence demonstrates how google_search and fetch_webpage are combined in practice: search to discover potential resources, then fetch specific pages to extract more detailed information.

3.4. Usage Patterns

Beyond individual tools, the evaluation examined how agents tend to sequence MCP-Web-Curl calls during problem solving. Several usage patterns emerged repeatedly in the logs.

In tasks that involve external documentation, the agent often starts with google_search to locate likely sources, then switches to fetch_webpage once a promising URL has been identified. When inspecting dynamic sites, the browser automation module waits for the page to settle before extracting text, which tends to yield stable content even when client side rendering is involved. In API-related tasks, the agent uses fetch_api to probe endpoints, varying payloads and headers across calls, and relies on status codes and sampled response bodies to infer the shape of the underlying service. When files are involved, download_file is typically invoked after either fetch_webpage or fetch_api has provided a direct link. The evaluation scenarios show that agents can autonomously fetch documentation, test endpoints, and download required artefacts without manual intervention [22].

These patterns are not enforced by the server itself. They arise from the combination of tool schemas, host prompts, and model behaviour. However, the consistency of the sequences across hosts and tasks suggests that MCP-Web-Curl provides a coherent set of capabilities that agents can recombine in different workflows.

3.5. Testing Case Without MCP-Web-Curl

To make the difference between environments with and without MCP-Web-Curl more concrete, a simple crypto price query was used as a small comparative test. In the first run, the agentic environment did not have access to

Jurnal Teknik Informatika Vol. 19 No. 1, April 2026 (122-134)
ISSN: p-ISSN 1979-9160 (Print)| e-ISSN 2549-7901 (Online)
DOI: https://doi.org/10.15408/jti.v19i1.49625

MCP-Web-Curl or any equivalent web tool. The user asked, in Indonesian, for the latest cryptocurrency prices. The system responded by switching from conversational mode to a code execution mode and issued a raw curl command to the Coingecko API in order to obtain live prices.

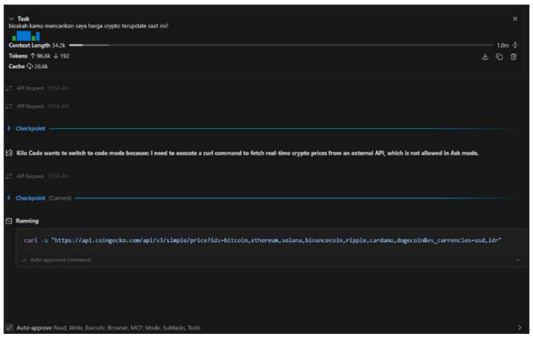

Figure 6. *Baseline Agent*

Figure 6. Shows the task description at the top, followed by an internal message where the agent explains that it needs to run a curl command because such calls are not allowed in the normal "Ask" mode. At the bottom, the environment displays the exact curl invocation. While this approach works, it breaks the abstraction of an agent that reasons through tools. Users must implicitly trust that the generated shell command is safe and, in many settings, such code execution would not be permitted at all.

3.6. Testing Case With MCP-Web-Curl

The same query was then repeated in the same environment, but this time MCP-Web-Curl was configured as an additional MCP server. The user prompt did not change. Internally, however, the agent chose a different strategy. Instead of attempting to run shell commands, it invoked the google_search tool provided by MCP-Web-Curl. The tool retrieved current price information from the web, and the agent produced a natural language answer in Indonesian listing approximate prices for several major cryptocurrencies.

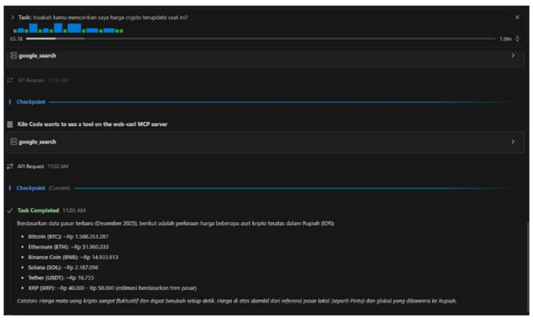

Figure 7. *MCP-Web-Curl Agent*

In Figure 7, the task description is identical to the previous case. The log shows a google_search call followed by an internal message indicating that "Kilo Code wants to use a tool on the web-curl MCP server". The task concludes with a regular textual response in the task panel, without any visible shell command or mode switch. From the user's perspective, the workflow remains purely conversational, yet it benefits from live web data. From a safety perspective, the environment no longer needs to grant direct access to arbitrary curl calls in order to satisfy such queries.

Taken together, Figures 6 and 7 illustrate how MCP-Web-Curl can shift an agentic environment from ad hoc code execution toward a more principled tool based interaction with external services.

3.7. Scenario Summary

Beyond the small crypto search example, the qualitative evaluation followed the three scenarios introduced in the methodology chapter: documentation reading, REST API inspection, and file download pipelines. In the documentation scenario, agents used google_search and fetch_webpage to locate and interpret version specific migration guides. This often led to updated code that matched the current version of a framework rather than the one implicitly encoded in model parameters. In the API inspection scenario, agents relied on repeated fetch_api calls to inspect responses, diagnose errors such as rate limiting or missing fields, and adjust client logic accordingly. In the file scenario, download_file was used to retrieve logs or datasets referenced in external pages and to pass the resulting file paths to host specific file reading tools.

The qualitative evidence from these scenarios suggests that MCP-Web-Curl can satisfy the functional requirements laid out in

Jurnal Teknik Informatika Vol. 19 No. 1, April 2026 (122-134)
ISSN: p-ISSN 1979-9160 (Print)| e-ISSN 2549-7901 (Online)
DOI: https://doi.org/10.15408/jti.v19i1.49625

Chapter 2 while remaining within the resource bounds expected for developer machines. The behaviour is not flawless, and agents occasionally misinterpret partial content or ignore truncation flags, yet the overall workflows look realistic and useful for everyday coding tasks.

3.8. Marketplace View

Internal trials are complemented by an external view drawn from MCP marketplaces. MCP-Web-Curl has been listed in public catalogues that curate MCP servers for various hosts. One such listing is shown in Figure 8.

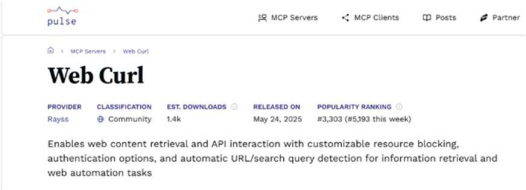

Figure 8. *Marketplace View*

The screenshot shows the Web Curl entry in the Pulse MCP server directory. The page identifies the provider, classifies the server as a community contribution, displays estimated download counts and a popularity ranking, and summarises the functionality as enabling web content retrieval and API interaction with configurable resource blocking and URL or query detection. Although such descriptions are brief, they indicate that MCP-Web-Curl is perceived as a general purpose web and API connector rather than a niche or experimental tool.

This marketplace representation aligns with the findings from the qualitative evaluation. In both views, MCP-Web-Curl appears as a reusable layer that connects agentic coding hosts to web pages and HTTP APIs, rather than as a monolithic agent or a one off integration.

3.9. Overall Discussion

The results in this chapter speak to the three research questions introduced in the introduction. Regarding the first question, the findings suggest that a lightweight MCP server with explicit tool schemas and modular internal components is a viable approach for providing a generic web and API layer for agentic coding environments. The architecture integrates cleanly with existing MCP hosts and can be deployed on ordinary developer hardware.

For the second question, the empirical material shows how tools such as google_search, fetch_webpage, fetch_api, and download_file map onto common workflows. Agents tend to compose these tools in consistent patterns, for example, search then fetch, repeated API probing with different payloads, and chaining retrieval and download operations. The Claude Desktop test and the crypto example make these patterns visible in concrete interfaces.

Addressing the third question, the broader implications of this design are evident. The comparison between environments with and without MCP-Web-Curl illustrates how a dedicated web and API layer can reduce reliance on raw code execution, make interactions more transparent, and encourage safer practices around external calls. The marketplace listing provides additional evidence that such a layer is valued by other practitioners and occupies a distinct position in the evolving ecosystem of MCP tools.

Qualitative analysis confirms that the MCP-Web-Curl artefact improves robustness and transparency compared to ad hoc web plugins, echoing results from design science evaluations of agentic AI frameworks in software engineering [23]. The comparison between environments with and without MCP-Web-Curl illustrates how a dedicated web and API layer can reduce reliance on raw code execution, make interactions more transparent, and encourage safer practices around external calls. Moreover, the observed performance stability across different hosts and settings resonates with broader evaluations of data science and statistics oriented agents that emphasise robustness of tool orchestration [24].

The overall results reinforce the idea that MCP-Web-Curl serves as a design-science artefact enabling transparent, repeatable, and verifiable agentic workflows, similar to findings from recent AI engineering case studies on trustworthy human–agent collaboration and multi-agent systems [25], [26].

CONCLUSION

This study has examined MCP-Web-Curl as a generic web and API access layer for MCP-based agentic coding environments. The study

concludes that MCP-Web-Curl demonstrates a viable implementation of the Design Science Research paradigm in AI tool engineering, providing an artefact that bridges reasoning and real-world web interaction [27]. This conclusion is consistent with current consensus that modular LLM-based multi-agent architectures are foundational for next-generation software engineering automation [28].

By treating MCP-Web-Curl as a design-science artefact, the work shows how a lightweight MCP server with carefully designed tool schemas can extend agentic coding assistants beyond their static training data. Agents equipped with MCP-Web-Curl can retrieve up-to-date documentation, probe live REST endpoints, and download auxiliary artefacts while keeping interactions observable and configurable for the host. The qualitative evaluation and marketplace evidence together indicate that such a web and API layer is both practically useful and conceptually aligned with emerging patterns in LLM-based agent design.

Future work will include integrating quantitative evaluations and larger user studies to complement the qualitative validation process [29]. Continued iterations should also align with emerging evaluation frameworks proposed for LLM-empowered agentic systems [30]. Ultimately, the MCP-Web-Curl artefact contributes to broader efforts in developing trustworthy and reproducible AI tools within the Design Science framework [24].

REFERENCES

[1] S. S. Chowa *et al.*, "From Language to Action: A Review of Large Language Models as Autonomous Agents and Tool Users," Oct. 2025, [Online]. Available: http://arxiv.org/abs/2508.17281

[2] H. Wang, J. Gong, H. Zhang, J. Xu, and Z. Wang, "AI Agentic Programming: A Survey of Techniques, Challenges, and Opportunities," Sep. 2025, [Online]. Available: http://arxiv.org/abs/2508.11126

[3] S. Barua, "Exploring Autonomous Agents through the Lens of Large Language Models: A Review," Apr. 2024, doi: 10.48550/arXiv.2404.04442.

[4] M. Haseeb, "Context Engineering for Multi-Agent LLM Code Assistants Using Elicit, NotebookLM, ChatGPT, and Claude Code," Aug. 2025, [Online]. Available: http://arxiv.org/abs/2508.08322

[5] R. Sapkota, K. I. Roumeliotis, and M. Karkee, "Vibe Coding vs. Agentic Coding: Fundamentals and Practical Implications of Agentic AI," May 2025, [Online]. Available: http://arxiv.org/abs/2505.19443

[6] S. I. Ross, F. Martinez, S. Houde, M. Muller, and J. D. Weisz, "The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development," in *Proceedings of the 28th International Conference on Intelligent User Interfaces*, New York, NY, USA: ACM, Mar. 2023, pp. 491–514. doi: 10.1145/3581641.3584037.

[7] C. Li *et al.*, "ModelScope-Agent: Building Your Customizable Agent System with Open-source Large Language Models," 2023. [Online]. Available: https://modelscope.cn/models

[8] C. Zhang *et al.*, "Large Language Model-Brained GUI Agents: A Survey," May 2025, [Online]. Available: http://arxiv.org/abs/2411.18279

[9] J. Cao and J. S. Rubin, "A Study on Deploying Large Language Models as Agents," 2024. Accessed: Dec. 10, 2025. [Online]. Available: https://dspace.mit.edu/bitstream/handle/1721.1/157177/cao-jiannan-sdm-sm-2024-thesis.pdf?sequence=1

[10] C. Berlin, "Exploring an AI Agentic Workflow for Solving Challenging Coding Problems: An Evaluation of a Large Language Model Based Multi-Agent System," 2022. Accessed: Dec. 10, 2025. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1949004/FULLTEXT02

[11] Y. Cai, R. Li, P. Liang, M. Shahin, and Z. Li, "Designing LLM-based Multi-Agent Systems for Software Engineering Tasks: Quality Attributes, Design Patterns and Rationale," Dec. 2025.

[12] J. He, C. Treude, and D. Lo, "LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 5, pp. 1–30, Jun. 2025, doi: 10.1145/3712003.

[13] H. Jin, L. Huang, H. Cai, J. Yan, B. Li, and H. Chen, "From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future," Apr. 2025.

[14] C. Rotar and Q. Zhang, "A design science research approach to Large Language Model-Based Agents for Requirements Specification (LLMBA4RS) in low-code applications," *Requir Eng*, vol. 30, no. 4, pp. 399–422, Dec. 2025, doi: 10.1007/s00766-025-00450-9.

[15] A. J. Asunmaa, "Large Language Models-Based Agents in Software Development," 2025.

[16] Lappeenranta-, "AGENT-BASED APPROACHES IN SOCIAL PROTOTYPING AND SOFTWARE DEVELOPMENT," 2025. Accessed: Dec. 11, 2025. [Online]. Available: https://lutpub.lut.fi/bitstream/handle/10024/170239/mastersthesis_gopalan_adhithyaraja.pdf

[17] S. P. Åström and S. Winoy, "Automating Software Development Processes Through Multi-Agent Systems: A Study in LLM-Based Software Engineering," 2024. Accessed: Dec. 11, 2025. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1871606/FULLTEXT02

[18] D. Hussein, "Usability of LLMs for Assisting Software Engineering: a Literature Review," 2024. Accessed: Dec. 11, 2025. [Online]. Available: https://elib.dlr.de/210605/1/Bachelorarbeit_Dania_Hussein___Systematisches_Literatur_Review__Eignung_von_ChatGPT_und_Co_als_Unterst%C3%BCtzung_beim_Software_Engineering.pdf

[19] M. Zijl, "Optimizing Software Engineering with Large Language Models: A Qualitative Study into the Integration of Generative AI," 2025, Accessed: Dec. 11, 2025. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1980230/FULLTEXT01.pdf

[20] A. M. Sami *et al.*, "System for systematic literature review using multiple AI agents: Concept and an empirical evaluation," Sep. 2025.

[21] M. A. Ferrag, N. Tihanyi, and M. Debbah, "From LLM Reasoning to Autonomous AI Agents: A Comprehensive Review," Apr. 2025.

[22] M. Sun *et al.*, "A Survey on Large Language Model-based Agents for Statistics and Data Science," *Am Stat*, pp. 1–14, Oct. 2025, doi: 10.1080/00031305.2025.2561140.

[23] J. Liu *et al.*, "Large Language Model-Based Agents for Software Engineering: A Survey," Dec. 2025.

[24] K. Chen, P. Wang, Y. Yu, X. Zhan, and H. Wang, "Large Language Model-based Data Science Agent: A Survey," Nov. 2025.

[25] K. Ronanki, "Facilitating Trustworthy Human-Agent Collaboration in LLM-based Multi-Agent System oriented Software Engineering," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, New York, NY, USA: ACM, Jun. 2025, pp. 1333–1337. doi: 10.1145/3696630.3728717.

[26] J. A. S. de Cerqueira, M. Agbese, R. Rousi, N. Xi, J. Hamari, and P. Abrahamsson, "Can We Trust AI Agents? A Case Study of an LLM-Based Multi-Agent System for Ethical AI," May 2025.

[27] S. Massoudi and M. Fuge, "Agentic Large Language Models for Conceptual Systems Engineering and Design," in *Volume 3B: 51st Design Automation Conference (DAC)*, American Society of Mechanical Engineers, Aug. 2025. doi: 10.1115/DETC2025-168856.

[28] R. Buchmann *et al.*, "Large language models: Expectations for semantics-driven systems engineering," *Data Knowl Eng*, vol. 152, p. 102324, Jul. 2024, doi: 10.1016/j.datak.2024.102324.

[29] C. Di Sipio, M. C. S. De Oliveira, D. Di Ruscio, P. T. Nguyen, and R. Rubei,

"Agentware in Software Engineering: A Taxonomy for Leveraging Llms-Based Multi-Agent Systems," 2025. doi: 10.2139/ssrn.5273078.

[30] S. Guo, C. Deng, Y. Wen, H. Chen, Y. Chang, and J. Wang, "DS-Agent: Automated Data Science by Empowering Large Language Models with Case-Based Reasoning," May 2024.