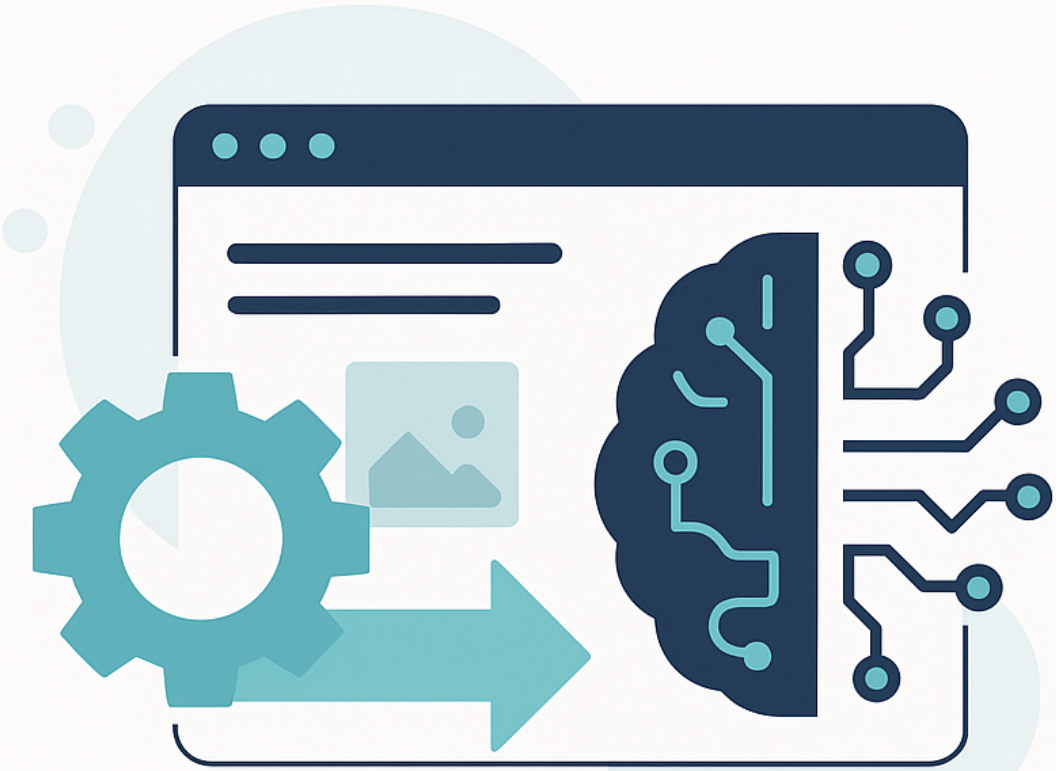
MCP–Selenium Bridge

A Generic Agentic Wrapper for Browser Automation

An Open Architecture for Agentic AI Systems
Independent of Proprietary Builders



White Paper

Prepared for: General Reader

Prepared by: Gene M. Arguelles, Consultant

October 31, 2025

MCP–Selenium Bridge: A Generic Agentic Wrapper for Browser Automation

An Open Architecture for Agentic AI Systems Independent of Proprietary Builders

Abstract

This paper introduces the MCP–Selenium Bridge, a generic agentic wrapper designed to enable browser automation as a first-class tool in Model Context Protocol (MCP)-compliant ecosystems. Unlike existing implementations bound to proprietary agent builders, this design allows any MCP client — including OpenAI's Agents SDK, Anthropic's Claude MCP, or independent agent controllers — to execute browser automation workflows through a FastAPI-based MCP server that wraps Selenium.

The paper presents a component-level architecture and process flow from user interface to back-end execution, illustrating how MCP standardization allows LLMs and agents to interact with real-world environments in a structured, verifiable, and context-aware manner.

1. Introduction

Large Language Models (LLMs) increasingly require structured mechanisms to interact with external systems. The Model Context Protocol (MCP) standardizes this interaction by defining a protocol through which models can discover, describe, and invoke external tools.

Browser automation — a foundational component of web interaction — remains largely siloed in frameworks like Selenium and Playwright. While these are powerful testing and automation engines, their integration with agentic frameworks remains under-developed.

The MCP–Selenium Bridge project aims to:

- Decouple browser automation from proprietary agent builders.
- Expose Selenium functionality as an MCP-compliant API.
- Provide a generic, reusable interface for agentic computing and AI evaluation workflows.

2. System Overview

The architecture consists of three major layers:

2.1 User Interface Layer

- Accepts user instructions in natural language or structured JSON.
- Sends requests to an MCP client (via Agents SDK or REST API).

2.2 MCP Wrapper Layer

- Implements FastAPI or Flask as an HTTP MCP server.
- Exposes standard endpoints: /mcp/schema, /mcp/invoke, /mcp/status.
- Serializes and deserializes JSON messages per the MCP specification.
- Invokes the appropriate backend tool (in this case, Selenium).

2.3 Backend Automation Layer (Selenium)

- Executes the requested browser automation tasks.
- Operates in headless mode for efficiency and scalability.
- Returns structured results (e.g., success status, page title, screenshots, logs).

3. Component Map

3. Component Map

A component map provides a high-level visualization of a system's architecture, illustrating how functional elements (or components) interact to achieve an end-to-end workflow. In the context of the MCP–Selenium Bridge, the component map shows the flow of information and control through five primary layers — each performing a distinct function in the process of transforming user intent into executable browser automation tasks and returning structured results to the user or calling agent.

The diagram highlights a top-down data and control flow, starting from the user's input interface and ending with a structured response produced by the system. Each layer abstracts specific logic, ensuring modularity, scalability, and easier debugging during integration with agentic AI systems.

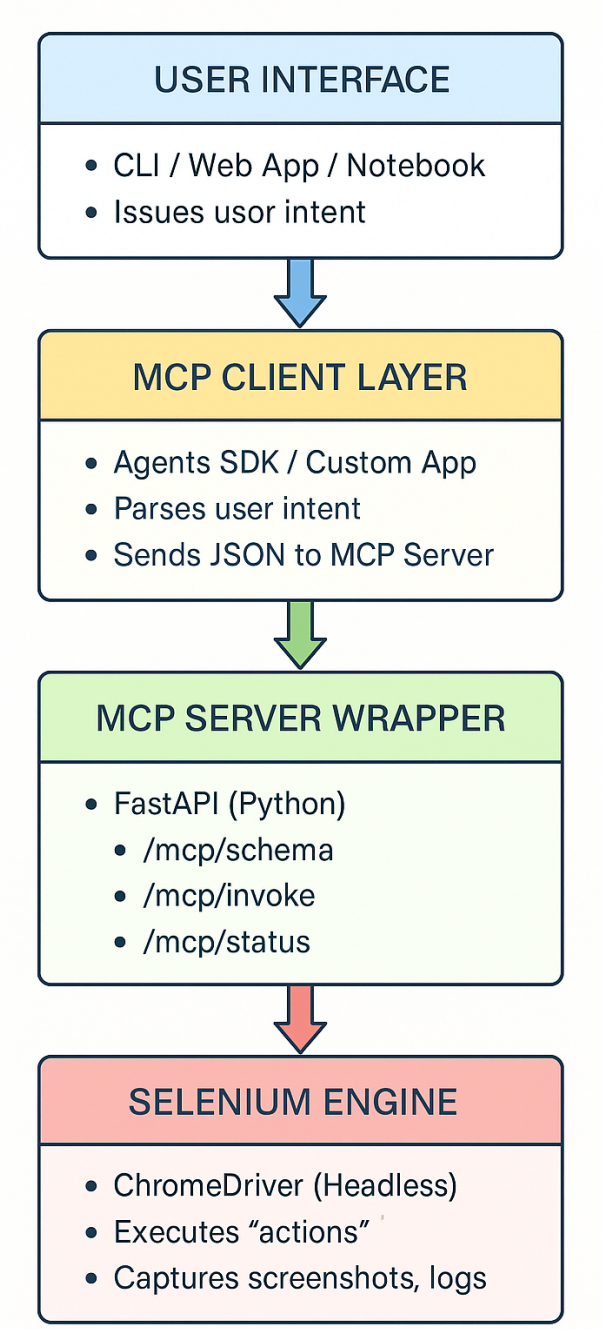


Figure 1. MCP–Selenium Bridge component map showing data flow across user, client, server, automation, and response layers.

1. User Interface Layer

This is the entry point of interaction between the user (or another agent) and the system.
It can take the form of:
• a Command-Line Interface (CLI) for developers,
• a Web Application for interactive dashboards, or
• a Jupyter Notebook or API client for testing.

The UI layer's role is to capture user intent — commands or natural language instructions — and pass them to the underlying agentic framework for interpretation.

———

2. MCP Client Layer

This layer acts as the interpreter between human-readable commands and machine-executable actions.
It uses the Agents SDK or a custom-built client that:

- Parses the user's intent,
- Structures the data into JSON following the Model Context Protocol (MCP) specification, and

- Sends it to the MCP Server Wrapper for execution.

This layer ensures that every instruction passed to the server is contextualized, validated, and protocol-compliant, making it a vital component for maintaining interoperability across agentic systems.

———

3. MCP Server Wrapper

The MCP Server Wrapper is the architectural heart of the system. It's built using FastAPI (Python) and provides the standardized MCP endpoints:
• /mcp/schema — advertises tool and parameter definitions,
• /mcp/invoke — handles tool execution requests, and
• /mcp/status — reports service health and availability.

This layer functions as a middleware translation hub, connecting the AI client's structured intent with the operational tools (in this case, Selenium). It ensures all communication adheres to the MCP JSON schema and translates incoming requests into executable backend calls.

———

4. Selenium Engine

At this layer, abstract instructions become concrete actions.
The Selenium Engine runs browser sessions (typically headless Chrome or Chromium) to:
- Visit specified URLs,
- Interact with web elements,
- Capture screenshots, and
- Log outputs or results.

It operates as the automation executor, allowing the MCP server to bridge LLM reasoning and real-world web interaction — effectively giving the agentic system a "hands-on" capability within web environments.

———

5. Response Processor

Once Selenium completes its operations, the Response Processor aggregates and formats the output.
It:
- Structures raw data (e.g., text, titles, screenshots, or logs) into standardized JSON,
- Adds metadata such as timestamps, status codes, or errors, and
- Returns the final result to the MCP Client Layer for presentation or further reasoning.

This final step ensures that the system's outputs are machine-readable, traceable, and ready for integration into multi-agent frameworks or audit workflows.

Summary

Together, these five layers form a cohesive agentic workflow that translates abstract AI reasoning into executable, testable browser actions.

The component map reflects the system's modular design philosophy, where each layer isolates a distinct responsibility — enhancing maintainability, extensibility, and compliance with the Model Context Protocol standard.

4. Process Map

1. User enters instruction (e.g., "Visit OpenAI.com and capture the homepage title").
2. MCP Client converts instruction into structured JSON payload.
3. MCP Server receives request via FastAPI, validates it.
4. Tool Invocation passes parameters to Selenium driver.
5. Backend Automation executes the action.
6. Response Aggregation collects results.
7. Response Serialization returns JSON response to client.
8. Display Layer presents interpreted results.

5. Technical Architecture

5.1 Tool Schema

The Tool Schema defines how the MCP server describes its available functions to clients or agentic systems. It acts as the contract of communication — a formal declaration of what tools exist, what each tool does, and what parameters are required to execute them.

In this example, the schema exposes a tool called selenium_browser, which enables AI agents or MCP clients to perform browser automation tasks. The schema specifies the tool name, description, and a JSON parameter object that includes properties such as "url", "action", and "selector". These parameters tell the MCP server what kind of automation to perform (e.g., visiting a page, clicking a button, or capturing a screenshot).

By conforming to the MCP specification, this schema ensures that any compliant client — whether OpenAI's Agent Builder, Anthropic's Claude MCP, or a custom-built SDK — can interpret and safely invoke the tool's capabilities without ambiguity.

```
{
  "mcp_version": "1.0",
  "tools": [
    {
      "name": "selenium_browser",
      "description": "Automate browser tasks via Selenium",
      "parameters": {
        "type": "object",
        "properties": {
          "url": {"type": "string"},
          "action": {"type": "string"},
          "selector": {"type": "string"}
        },
        "required": ["url", "action"]
      }
    }
```

Figure 2: Tool schema example

5.2 MCP Invocation Request

The MCP Invocation Request represents the structured message sent by the client to the MCP server to execute a specific tool defined in the schema.

This JSON payload acts as the execution command within the MCP workflow, containing two essential elements: the "tool" identifier (which references a declared tool in the schema) and the "parameters" object (which carries the input values for the operation).

In the example provided, the request instructs the server to execute the "selenium_browser" tool and perform the "visit" action on "https://openai.com". This standardized format allows clients and agents to communicate intent without relying on natural language interpretation — promoting deterministic, testable, and secure automation behavior.

The invocation structure also enables error handling and validation, allowing the MCP server to verify input conformity before triggering any browser automation routines, reducing the risk of malformed requests or unintended actions.

```
{
  "tool": "selenium_browser",
  "parameters": {
    "url": "https://openai.com",
    "action": "visit"
  }
}
```

Figure 3: Invocation Request example

5.3 Response

The Response Example below illustrates the output structure returned by the MCP server after successful tool execution.

Here, the server replies with a JSON object containing two key elements: "status" and "output". The "status" field indicates whether the operation completed successfully or failed, while "output" holds the structured data resulting from the operation — in this case, a captured "page_title" and an execution "timestamp".

This format transforms what would otherwise be unstructured execution logs into a machine-readable, traceable record of system behavior. It enables downstream agents, evaluators, or analytics systems to parse and interpret results consistently across different tool implementations.

The standardized response model is fundamental to agentic interoperability, ensuring that multiple tools — even across different domains — can report results in a uniform, schema-compliant way. This not only aids debugging and reproducibility but also supports future extensions such as evaluation scoring, compliance logging, or chained multi-agent orchestration.

```
{
  "status": "success",
  "output": {
    "page_title": "OpenAI – Discover the Future",
    "timestamp": "2025-10-29T10:45:00Z"
  }
}
```

Figure 4: Status Response example

6. Implementation Details

Frontend: Python CLI / Web UI / Agent SDK
MCP Wrapper: FastAPI + JSON-RPC
Automation Engine: Selenium WebDriver (Headless)
Data Logging: SQLite / JSON / Loguru
Security Layer: API key auth, input sanitization

The implementation of the MCP–Selenium Bridge integrates multiple modular layers, each responsible for a specific operational domain within the system. The Frontend may be realized as a Python Command-Line Interface (CLI), a lightweight Web UI, or an Agent SDK client — all serving as entry points for issuing structured instructions to the MCP server. The MCP Wrapper, built using FastAPI and supporting JSON-RPC, functions as the communication backbone, handling schema discovery, invocation routing, and response delivery in compliance with the Model Context Protocol. The Automation Engine, powered by Selenium WebDriver running in headless mode, executes browser-based tasks such as navigation, interaction, and content capture without requiring a visible browser window. Data Logging is managed through SQLite for persistence, JSON for structured interchange, and Loguru for event tracking and debugging, ensuring transparent observability across runs. Finally, the Security Layer enforces API key–based authentication and input sanitization, protecting the system from unauthorized access and command injection vulnerabilities. Collectively, these components form a cohesive, secure, and extensible foundation for agentic browser automation and AI-driven workflow execution.

7. Advantages

Below is a list of some of the potential advantages identified:

- Generic & Portable
- Decoupled from Agent Builder
- Bridges LLM reasoning with browser automation
- Open-Source Ready
- Extensible Design

The MCP–Selenium Bridge introduces several key advantages that make it a versatile and forward-looking framework for agentic AI development. First and foremost, it is generic and portable, meaning it adheres to open standards rather than vendor-specific APIs. Any client or agent capable of interpreting the Model Context Protocol can interact with the system, enabling seamless integration across diverse ecosystems such as OpenAI's Agent SDK, Anthropic's Claude MCP, or future agentic frameworks. This openness ensures long-term adaptability and minimizes the risk of technological lock-in.

Equally important, the design is decoupled from Agent Builder, allowing developers to deploy and test MCP-compliant workflows independently of proprietary tooling. This architectural independence expands the experimental and deployment scope—teams can embed the bridge into local development environments, research sandboxes, or enterprise production pipelines without dependency conflicts. By bridging LLM reasoning with browser automation, the system transforms abstract AI instructions into tangible, verifiable actions, effectively giving language models the ability to perceive and manipulate the web environment in a controlled, auditable manner.

Furthermore, the project is open-source ready, making it ideal for community contribution, peer review, and rapid innovation. Its extensible design allows new tools, endpoints, and automation modules to be added without restructuring the core framework. Each layer—from schema definition to invocation handling—can evolve independently, facilitating continuous improvement. Together, these attributes establish the MCP–Selenium Bridge as a foundational reference model for next-generation, interoperable, and transparent agentic systems.

8. Potential Use Cases

Below is a list of some potential Use Cases identified:

- Automated Web QA via LLMs
- AI-Powered Compliance Bots
- Creative Automation
- Research & Monitoring

The MCP–Selenium Bridge opens a wide spectrum of applications that merge language model reasoning with browser-level automation, enabling practical, scalable deployments of agentic AI in real-world environments. Because it abstracts web automation into a standardized MCP interface, developers and researchers can adapt it for diverse domains ranging from quality assurance and regulatory compliance to creative production and intelligence gathering.

One of the most immediate applications is Automated Web Quality Assurance (QA) via LLMs. Traditional web testing frameworks rely heavily on human-authored scripts to verify functionality, accessibility, and visual integrity. By integrating the MCP–Selenium Bridge, an LLM can autonomously interpret test instructions, trigger browser actions, and analyze rendered results, all while maintaining a structured feedback loop. This allows for dynamic QA automation, where the agent can reason about errors, suggest remediations, and continuously refine its test coverage — effectively combining human-level insight with machine-scale execution.

A second domain involves AI-Powered Compliance Bots, where regulatory or ethical requirements (such as FDA, GDPR, or ISO standards) must be verified across web content or enterprise platforms. By leveraging Selenium through the MCP interface, an AI compliance agent can systematically inspect documentation, capture evidence, and generate structured compliance reports. This bridges the gap between static AI analysis and verifiable action-based auditing, ensuring that regulatory adherence is both measurable and reproducible.

The system also lends itself to Creative Automation, particularly in the intersection of digital storytelling, marketing, and media production. Here, agentic systems can interact with creative platforms to generate, evaluate, and refine content dynamically. For example, an AI writer or designer could use the MCP–Selenium Bridge to interface with design tools or content management systems, automatically publishing updates or optimizing layout configurations based on contextual cues. This transforms creativity from a purely generative process into a cybernetic loop of creation, evaluation, and iteration.

Lastly, the framework is ideally suited for Research and Monitoring applications that require persistent, structured observation of digital ecosystems. The MCP–Selenium Bridge allows agents to autonomously visit websites, extract data, and summarize findings using natural language reasoning. This is valuable in fields such as market intelligence, cybersecurity, and trend analysis, where AI-driven agents can continuously monitor environments and produce human-readable reports. When combined with contextual reasoning, this capability transforms routine data collection into adaptive knowledge synthesis — providing organizations with timely, actionable insights.

In essence, these use cases highlight how the MCP–Selenium Bridge acts as a conduit between AI cognition and digital action, offering a unifying infrastructure for automation, compliance, creativity, and research. By translating intent into interaction, it not only expands what AI systems can observe and execute but also sets the stage for more autonomous, transparent, and collaborative AI operations in the future.

9. Future Work

Below is a list of identified items to consider for future expansion and development"
- Extend schema for multi-step browser sessions.
- Integrate Playwright as alternative backend.
- Add evaluation layer for scoring success.
- Publish as open-source reference.
- Develop GUI dashboard for visualization.

The evolution of the MCP–Selenium Bridge provides a strong foundation for agentic browser automation, but its full potential lies in the continued expansion of its capabilities. Future development will focus on enhancing multi-step reasoning, expanding backend interoperability, strengthening evaluation mechanisms, and improving user accessibility through visualization and open collaboration. These efforts will push the framework closer to a production-grade reference implementation for standardized, agent-driven automation systems.

The first area of advancement is the extension of the MCP schema to support multi-step browser sessions. Presently, the bridge operates in a single-action mode, where each invocation executes one discrete browser task. By evolving the schema to accommodate session persistence and chained interactions, an agent could maintain browser context across multiple operations — such as logging in, navigating through a workflow, collecting data, and submitting results. This enhancement would move the architecture from a stateless model toward a context-retaining automation engine, aligning it more closely with real-world user journeys.

A second priority is to integrate Playwright as an alternative backend to Selenium. Playwright offers parallel browser instances, faster execution, and broader cross-browser compatibility. By abstracting the automation layer beneath a common MCP interface, the framework can dynamically select between Selenium and Playwright at runtime based on task requirements. This modularity ensures greater performance flexibility and future-proofs the bridge against changes in browser automation ecosystems.

The third planned enhancement is the introduction of an evaluation layer for success scoring and performance analytics. This layer would quantify the effectiveness of automation actions by tracking metrics such as completion rate, latency, and semantic accuracy of task fulfillment. Integrating scoring mechanisms would transform the bridge from a purely functional interface into an AI evaluation and learning platform, where agents can iteratively refine their behaviors based on feedback and benchmarking results.

Equally important is the intent to publish the MCP–Selenium Bridge as an open-source reference implementation. Open sourcing will encourage community participation, independent audits, and interoperability testing, accelerating the protocol's adoption and maturation. The repository can serve as a canonical model for MCP-compliant tool design, inspiring extensions into domains such as API testing, robotics control, and document evaluation.

Finally, development will include a graphical dashboard for monitoring and visualization. The GUI will allow users to observe active MCP sessions, view execution logs, inspect JSON exchanges, and

analyze performance metrics in real time. This human-centric interface will bridge interpretability and usability, providing both engineers and researchers with transparent oversight of agentic workflows.

Collectively, these future directions aim to strengthen the scalability, transparency, and usability of the MCP–Selenium Bridge. As the project evolves, it will not only serve as a technical framework but also as a research and educational reference for the emerging discipline of Agentic Computing — demonstrating how open protocols can unify intelligent reasoning and autonomous digital action.

10. Conclusion

The MCP–Selenium Bridge demonstrates how open protocols can unify disparate AI tools into coherent, actionable systems. By exposing browser automation via a generic MCP interface, this project sets a precedent for building agentic, testable, and transparent AI ecosystems that function beyond proprietary silos.

The MCP–Selenium Bridge represents a pivotal step toward realizing the promise of Agentic Computing — the convergence of intelligent reasoning systems with executable automation frameworks. By formalizing communication through the Model Context Protocol (MCP), this project demonstrates how disparate technologies such as large language models, browser automation engines, and web APIs can interoperate seamlessly within a common protocol structure. The result is a transparent, verifiable, and extensible system where language models can move beyond abstract reasoning to perform grounded, measurable actions in real-world environments.

At its core, the bridge exemplifies a design philosophy of modularity and interoperability. Each layer — from the user interface and MCP client to the Selenium execution engine and response processor — performs a distinct function within an orchestrated workflow. This separation of concerns ensures that the system remains both adaptable and maintainable, capable of evolving alongside emerging AI standards and automation technologies. By decoupling the framework from proprietary builders, the project establishes a foundation for open experimentation and cross-platform compatibility, fostering an ecosystem of agentic systems that can communicate and collaborate across organizational and technological boundaries.

Beyond its immediate technical value, the MCP–Selenium Bridge contributes to a broader conversation about AI alignment, transparency, and accountability. In an era where autonomous agents are increasingly tasked with decision-making and execution, maintaining a clear, auditable chain of logic between intent and action is essential. Through its structured invocation schema, standardized responses, and modular architecture, this framework offers a reproducible pathway toward explainable automation — where every action performed by an agent can be traced, validated, and improved.

Looking forward, the MCP–Selenium Bridge stands as both a reference implementation and a conceptual model for how future agentic systems can operate. As it matures into an open-source, community-driven platform, its impact will extend beyond browser automation to encompass other domains — from software validation and compliance testing to creative collaboration and research intelligence. The project underscores a central principle of the agentic paradigm: when context, protocol, and action are unified, intelligence becomes operational.

In summary, this work marks an important milestone in the evolution of interoperable AI architectures. The MCP–Selenium Bridge is not just a toolset but a blueprint for how next-generation AI systems can think, act, and learn within structured, transparent, and human-aligned boundaries — advancing the

frontier of agentic automation and establishing the groundwork for a truly collaborative future between humans and machines.