

# AI-Powered Travel Planning with Model Context Protocol (MCP): A Survey

Sarvesh Salunkhe, Yashraj Kadam, Ajay Jadhav, Nikhil Vhorkate, Mr. Pradeep.M.Patil

Vidya Pratishthan's Kamalnayan Bajaj Institute of Engineering and Technology, Baramati, Maharashtra

**Abstract**—Large Language Models have become indispensable for modern AI systems, but their effectiveness in practice is limited by the complex rules for stateless integration. This phenomenon results in a so-called “disconnected models problem”: An AI agent controlled by an LLM is unable to keep continuity over more than one action or task. Most of the state-of-the-art systems demonstrate this flaw by utilising custom integration logic, typically called “glue code”. While such a solution can be effective in a small scale, it does not scale well and is subject to security vulnerabilities and maintenance challenges.

To address these challenges, the Model Context Protocol (MCP) has been proposed as a standardised interface for connecting AI models to external tools and data. The MCP follows a session-oriented architecture using JSON-RPC to enable more consistent and stateful interactions between systems

In this survey, we examine the architecture of MCP, focusing on its host–client–server model and its main primitives, namely tools, resources, and prompts. We also present “TripSync,” a practical case study that shows how an MCP-based travel planner can coordinate multiple travel-related microservices within a single system.

In addition, we compare MCP with other emerging agent communication protocols, including Agent Communication Protocol (ACP), Agent-to-Agent (A2A), and Agent Network Protocol (ANP). Based on recent studies of more than 8,000 MCP servers, we discuss the current state of the ecosystem and highlight important security concerns such as tool poisoning and credential exposure. Finally, we outline the potential role of MCP in building modular, secure, and context-aware AI systems in the future.

**Index Terms**—Model Context Protocol (MCP), Large Language Models (LLMs), Agent Interoperability, Tool Poisoning, AI Security, Gemini API

## I. INTRODUCTION

Large Language Models (LLMs) such as Google’s Gemini and Anthropic’s Claude have become an important part of modern artificial intelligence. They are widely used to power autonomous agents that can operate in different environments [16]. However, their practical usefulness is still limited because they mainly depend on static training data and cannot directly interact with real-time external systems [1], [7]. This dependence on internal knowledge creates a major bottleneck when building real-world applications.

Notably, this limitation is evident in travel planning. Recent work, such as the “Full Stack AI Trip Planner”, leveraging React and Firebase, can generate personalised itineraries [13]. However, integrating such systems with multiple services involves developing custom logic on top of the basic system that is not standardized. A prime example for our use case is

the “GPT Trip Planner” that can similarly generate context-aware itineraries [14]. But whenever the user preferences and constraints change, the entire itinerary will have to be regenerated. Building such systems typically requires writing custom integration logic – commonly referred to as “glue code” [9]. Although “glue code” can solve the immediate problem, it can’t scale and is difficult to secure and maintain in the long run [1], [7]. This software design problem underscores the need for standardisation in such systems.

In late 2024, Anthropic introduced the Model Context Protocol (MCP), a general-purpose protocol standardizing AI-tool interactions [1], [6], [15]. Inspired by the Language Server Protocol (LSP), MCP provides a universal, model-agnostic framework for AI applications to communicate with external tools dynamically [1]. Instead of relying on predefined tool mappings, MCP allows AI agents to autonomously discover, select, and orchestrate tools based on task context [7]. The rapid emergence of a community-driven ecosystem, with thousands of community-driven MCP servers enabling access to systems like GitHub and Slack, underscores its growing importance [1], [5].

This paper provides a consolidated survey of the MCP ecosystem. We detail its architecture, compare it with other emerging protocols, and conduct an in-depth analysis of its security threats. We use a practical travel planning application, “TripSync,” as a case study to demonstrate how MCP and the Gemini API can be integrated to solve complex, real-world tasks. This approach contrasts with other generative AI planners that use monolithic stacks like React and Firebase [13].

## II. THE LANDSCAPE OF AGENT COMMUNICATION

Before MCP, AI applications relied on various methods to interact with external tools, each with significant limitations that created the need for a standardized protocol.

### A. From Manual Wiring to Standardized Plugins

In the early stages, developers mainly depended on “manual API wiring,” commonly known as “glue code,” to connect AI models with external tools. This approach required custom handling for authentication, data conversion, and error management for every single integration [9]. As the number of models and tools increased, this created an  $M \times N$  integration problem, where  $M$  different models had to be connected with  $N$  different tools [7]. Such systems became tightly coupled

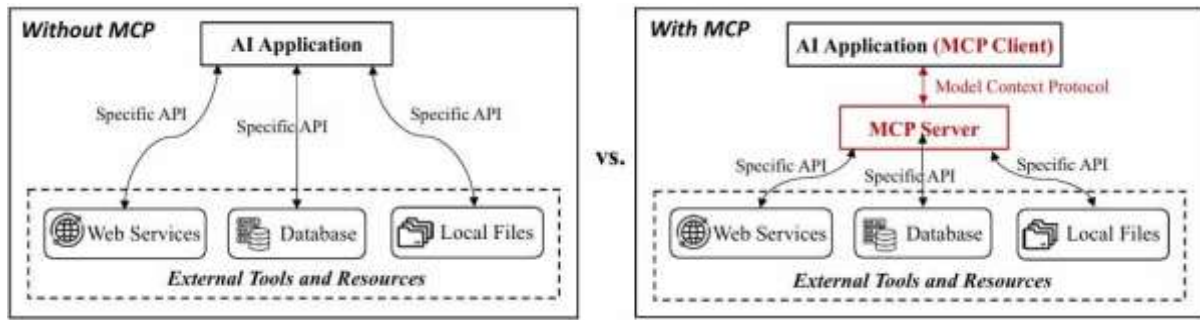


Fig. 1. Tool invocation with and without MCP [1].

and fragile, making them difficult to scale or maintain. For example, a Stripe payment tool designed for one framework, such as LangChain, would not work directly with another framework like LlamaIndex. As a result, developers often had to rewrite the same integration logic multiple times [4].

To reduce this complexity, standardized plugin interfaces were introduced. One example is OpenAI's ChatGPT Plugins, which allowed AI models to connect with external services using structured schemas like OpenAPI. Although this was an improvement over manual wiring, these solutions were usually platform-specific and supported only one-directional communication, leading to isolated ecosystems rather than a truly interoperable system [1].

### B. Function Calling and its Limitations

The introduction of "function calling" by OpenAI in 2023 marked an important step forward, as it allowed LLMs to interact with external APIs in a structured manner [18]. Frameworks such as LangChain were built on top of this idea to support more advanced agent-like behavior. However, this approach still has several important limitations [2], [6]:

- **Static Definitions:** In most implementations, tools must be defined in advance within the prompt. This means the model cannot dynamically discover or use new tools at runtime.
- **Ad-hoc Security:** Security mechanisms are usually specific to each framework and are not standardized. As a result, token handling and access control can be inconsistent, and the system may be vulnerable to issues such as tool poisoning caused by malicious inputs [4], [8].
- **Context Limitations:** These systems are generally stateless. The model needs to be reminded of the available tools and the conversation history at every step, which increases overhead. This contributes to the "disconnected models problem" [3], where important context must be repeatedly transmitted within the limited context window.
- **Platform Lock-in:** Tools designed for one provider's function-calling format are often not compatible with others. For example, a tool built for OpenAI may not work directly with systems from Google or Anthropic, leading to a fragmented ecosystem [2].

Approaches such as ReAct [20] and Toolformer [19] improved the ability of models to reason and act, but they still did not fully address the broader interoperability problem.

### C. The Need for a Protocol Standard

These challenges highlight the urgent need for a protocol-level standard to unify knowledge grounding with action invocation. The ad-hoc, stateless nature of function calling is insufficient for building robust, stateful, and secure AI agents [6]. What was needed was a "USB-C for AI" [5], [6]—a universal, model-agnostic, and secure protocol.

### D. MCP as the Solution

MCP was designed specifically to overcome these limitations. It addresses the "disconnected models problem" by providing a standardized, session-oriented framework [3], [6].

- **Standardized & Interoperable:** As a protocol, MCP is model-agnostic. An MCP server for Stripe can be built once and used by any MCP-compliant client, whether it's powered by Gemini, Claude, or GPT-4 [4].
- **Dynamic Discovery:** The MCP client-server architecture allows an AI application (the "Host") to dynamically discover, connect to, and negotiate capabilities with tools (the "Servers") at runtime [7].
- **Stateful & Session-Oriented:** MCP is built on JSON-RPC 2.0 and supports persistent, stateful sessions. This solves the "disconnected models problem" by allowing context to be maintained across multiple interactions without resending the entire history [3], [6].
- **Security-First Design:** MCP integrates security as a core primitive. It uses modern standards like OAuth 2.1 and PKCE for authorization, managing tokens and ensuring that agents have fine-grained permissions [6], [8].

## III. MODEL CONTEXT PROTOCOL (MCP) ARCHITECTURE

The TripSync system uses the MCP architecture to connect the frontend with the backend services and the AI planning components. In this section, we first explain the general MCP architecture and then describe our case study in detail.

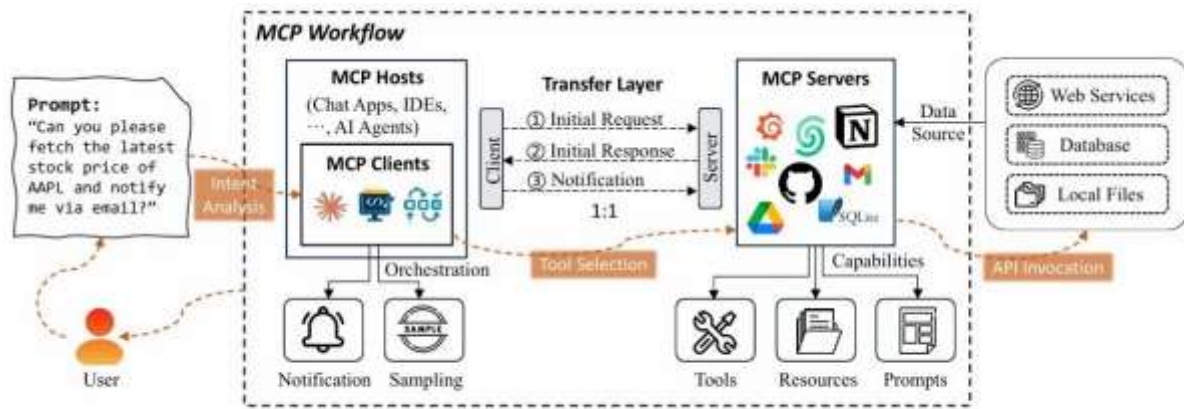


Fig. 2. The workflow of MCP.

Fig. 2. The general workflow of the Model Context Protocol (MCP), showing the Host, Client, and Server components and their interactions via JSON-RPC over different transports [1].

### A. The MCP Architecture In Depth

The MCP protocol uses a client-server model to define a standard way in which an AI application communicates with external tools and data sources [7]. The architecture of the protocol is illustrated in Fig. 2, which consists of three major components. These components interact to provide a smooth communication protocol on a transport layer that adheres to the JSON-RPC 2.0 specification [1], [6].

1) *MCP Host*: The MCP host is the main AI application that provides the environment where tasks are executed. Examples include our “TripSync” FastAPI backend, an IDE assistant like Cursor, or chat-based applications such as Claude Desktop [1], [7]. The host is responsible for managing client sessions, maintaining context, and coordinating with the LLM [3], [6]. It also handles privacy settings, permission control, and sandboxing policies [6]. A single host can connect to multiple MCP servers at the same time, which makes it suitable for complex, multi-step workflows [7].

2) *MCP Client*: The MCP client acts as an intermediary inside the host. Its main role is to manage the communication with a specific MCP server [7]. Each client represents an independent session [6]. It sends requests, receives responses or notifications, and ensures secure data exchange between the host and the server [1]. In our case study, the FastAPI backend creates separate MCP client instances for each travel-related microservice, such as flights, hotels, and trains.

3) *MCP Server*: The MCP server is a program that provides access to external systems by exposing capabilities such as tools, resources, and prompts [1], [7]. These servers can run locally using standard input/output communication or remotely using streamable HTTP with Server-Sent Events, depending on the deployment requirements [5], [6]. In the TripSync project, `flight_server.py`, `hotel_server.py`, and `train_server.py` act as individual MCP servers for their respective domains.

### B. Core Capabilities (Primitives)

The MCP Server exposes its functionalities through four core primitives, each with a distinct control model [6], [7]:

- **Tools (Model-Controlled)**: Executable functions the LLM can invoke to perform specific actions, such as `search_flights` or Google Hotels. The server advertises its tools, and the LLM (via the host) decides when to call them [4], [6]. This is the most common primitive for agentic action.
- **Resources (Application-Controlled)**: Data elements exposed to the application, such as files, database entries, or conversation history [7]. The client application, not the LLM, manages and retrieves these. This is ideal for providing large volumes of passive context (e.g., RAG) to the host [6].
- **Prompts (User-Controlled)**: Reusable templates for common interactions designed to guide the AI model [4]. This allows a user to trigger a predefined, complex workflow (e.g., ‘summarize this file’) [6].
- **Sampling (Server-Controlled)**: An advanced primitive that allows the “server” to delegate LLM text generation “to the client” for fine-grained control [6]. This reverses the typical flow of control and is used in specialized cases.

### C. MCP Communication Workflow and Lifecycle

MCP standardizes the entire session lifecycle, which is one of its main advantages over stateless function-calling approaches [6]. The protocol is built on JSON-RPC 2.0, a lightweight remote procedure call format that defines how requests and responses are structured [2], [6]. The lifecycle of an MCP session can be divided into three main phases [1], [6]:

- 1) **Initialization Phase**: The session begins when the client sends an `initialize` request to the server. During this step, both sides exchange information about their

supported protocol versions and capabilities, such as whether they support tools or sampling. This initial handshake ensures that the client and server are compatible before any actual tasks are performed [6].

- 2) **Operation Phase:** After initialization, the client and server start exchanging regular protocol messages. These JSON-RPC requests and responses are used to execute tools, access resources, or send progress updates. This phase is stateful and can remain active for a long time, depending on the task [1], [6].
- 3) **Shutdown Phase:** When the session is no longer required, the connection is closed. This usually happens by terminating the transport channel, such as closing the stdio pipe or the HTTP connection. Both the client and server are then expected to release any resources related to that session [1], [6].

Because the session remains stateful, MCP can maintain context across multiple interactions without repeatedly sending the full conversation history. This helps address the “disconnected models problem” [3].

MCP supports two main transport methods for sending JSON-RPC messages [6]:

- **Stdio:** This method is used for local communication, where the host starts the MCP server as a subprocess. It is commonly used in desktop applications or IDE plugins and provides low latency [6].
- **Streamable HTTP + SSE:** This method is used for remote servers. The client sends POST requests for standard operations, while a persistent GET connection receives Server-Sent Events (SSE) for asynchronous updates such as progress notifications [6].

#### D. Case Study: The “TripSync” MCP-Based Travel Planner

To illustrate the real-world applicability of Model Context Protocol, we created a prototype system, *TripSync*. TripSync is an AI-enabled travel planner that leverages MCP to orchestrate multiple travel services and move from an end-user’s single natural language request to a complete itinerary.

1) **System Overview:** TripSync follows a modular architecture built on MCP. The system is composed of the following main components:

- **Frontend (React Interface):** A web interface where users can enter travel details such as destination, dates, budget, and preferences.
- **MCP Host (FastAPI Backend):** Acts as the central controller. It receives the user request and manages communication with all services.
- **Planner Service:** Interprets the user query and decides which travel services are needed.
- **Supervisor Agent:** Coordinates all travel-related services such as flight, hotel, budget, and places servers. It manages parallel communication between services and combines their responses to generate an efficient travel plan.
- **MCP Client:** Creates communication sessions with the required MCP servers using the MCP protocol.

#### • MCP Servers:

- Flight MCP Server
- Hotel MCP Server
- Train MCP Server
- Bus MCP Server
- Places MCP Server
- Budget MCP Server

Each server provides specific tools, such as searching for flights, checking hotel availability, or retrieving train options.

- **AI Service (LLM):** Processes the collected travel data and generates one or more optimized itineraries based on the user’s preferences.

2) **Workflow:** The overall operation of the system can be described in the following steps:

- 1) The user submits a natural language request, for example, “Plan a 3-day budget trip to Goa.”
- 2) The request is sent to the FastAPI backend, which acts as the MCP host.
- 3) The planner service analyzes the request and identifies the required travel services, such as flights, hotels, or trains.
- 4) The MCP client establishes sessions with the corresponding MCP servers.
- 5) Each MCP server executes its tools (for example, `search_flights` or `search_hotels`) and returns structured results.
- 6) The host collects and combines the results from all servers.
- 7) The aggregated data is passed to the LLM, which generates one or more optimized itineraries.
- 8) The final travel plan is returned to the frontend and displayed to the user.

3) **Key Architectural Benefits:** The TripSync implementation highlights several advantages of using MCP:

- **Modularity:** Each travel service runs as a separate MCP server, making the system easier to manage.
- **Scalability:** New services, such as bus booking or activity recommendations, can be added without major changes to the host.
- **Interoperability:** The host communicates with all services through a single standardized protocol.
- **Dynamic orchestration:** The planner service selects tools based on the user’s request, allowing flexible and context-aware behavior.

This case study shows how MCP can support a decoupled, scalable, and context-aware architecture for practical AI applications such as travel planning [6], [7].

#### IV. COMPARATIVE ANALYSIS WITH OTHER PROTOCOLS

MCP is one of several emerging protocols that aim to improve interoperability between AI agents and external services. Table (1) presents a comparison of MCP with other protocols such as Agent Communication Protocol (ACP), Agent-to-Agent Protocol (A2A), and Agent Network Protocol (ANP),

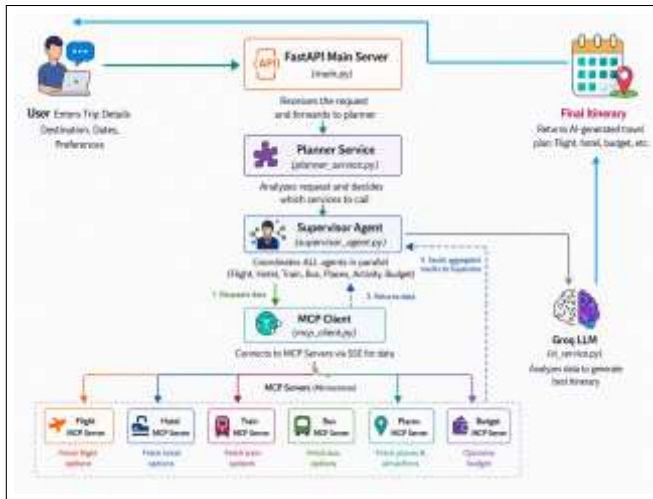


Fig. 3. Architecture of the TripSync MCP-based travel planning system, showing the flow from the frontend to the FastAPI host, planner service, MCP client, domain-specific MCP servers, and the LLM-based AI service.

highlighting their different architectures and intended use cases [2], [9].

MCP mainly focuses on “vertical” integration, where a central LLM host connects directly to its external tools and services [9]. In contrast, protocols like A2A are designed for “horizontal” communication, which involves collaboration between multiple agents. For example, a planning agent may delegate a task to a separate booking agent [2], [9]. A2A uses the concept of “Agent Cards” to help agents advertise their capabilities in a decentralized manner, which is different from MCP’s direct capability negotiation through a client–server handshake [2].

For our travel planning case study, MCP is a suitable choice because it is specifically designed to standardize the interaction between a central AI planner and multiple external travel services.

## V. SECURITY ANALYSIS OF THE MCP LIFECYCLE

MCP servers are open and extensible by design, which makes them flexible but also introduces several security risks throughout their lifecycle [1]. Maintaining strong security practices is especially important for applications like “Trip-Sync,” which may eventually handle sensitive information such as user data and payments. In this section, we analyze the major threats across three phases: creation, operation, and update [1], [8].

### A. Security Risks in the Creation Phase

The creation phase includes tasks such as registering the server, installing it, and verifying the integrity of its code. Several risks can arise during this stage [1]:

1) *Name Collision*: In this scenario, a malicious actor registers an MCP server with a name that is identical or very similar to a legitimate one, such as “Stripe-MCP” instead of “Stripe\_MCP.” This can mislead users and result in impersonation attacks [1].

2) *Installer Spoofing*: Attackers may distribute modified installers that contain hidden malicious code or backdoors. Unofficial auto-installers, such as mcp-get, may provide convenience but can also become entry points for such attacks [1].

3) *Code Injection/Backdoor*: Malicious code can also be introduced directly into the server’s source code or its dependencies. For example, a compromised helper library may include hidden backdoors, creating long-term supply-chain risks [1].

### B. Security Risks in the Operation Phase

The operation phase is when the server actively processes requests and executes tools. This is considered the most critical phase from a security perspective, especially for a travel planning application. Some of the key risks are described below.

1) *Tool Poisoning*: Tool poisoning is a serious threat in which an attacker uses a carefully crafted prompt to trick the LLM into calling a tool in an unintended or harmful way [8]. For example, a malicious query could instruct the system to ignore previous instructions and use a booking tool with manipulated parameters, such as booking a flight to a fake destination at an extremely low price [4].

2) *Command Injection via Prompt Semantics*: In this attack, a user provides input that is inserted into a prompt template. The LLM may interpret part of this input as a privileged command, which can lead to data leakage or unauthorized actions [6].

3) *Tool Name Conflicts and Hijacking*: If multiple tools have similar or identical names, the system may become confused about which one to use. Attackers can exploit this by adding misleading phrases to a tool’s description, such as instructions that make the tool appear more important or trustworthy than others [1].

4) *Sandbox Escape*: Some tools run code in a restricted environment, often called a sandbox. If there are weaknesses in this sandbox, an attacker may be able to escape the restricted environment and gain access to the host system [1].

5) *SSE Channel Exhaustion*: For MCP servers that use Streamable HTTP, attackers may open a large number of Server-Sent Event (SSE) connections and keep them active. This can consume server resources such as memory and network sockets, leading to denial-of-service conditions [6].

### C. Security Risks in the Update Phase

The update phase involves managing new versions of the server and its configuration. Several risks may occur during this stage [1]:

1) *Privilege Persistence*: Sometimes, outdated or revoked permissions remain active even after an update. This can allow unauthorised access to continue unnoticed [1].

2) *Re-deployment of Vulnerable Versions*: Because MCP is decentralised, there is no single authority to verify server versions. As a result, users may accidentally install or revert to older versions that contain known vulnerabilities [1].

TABLE I  
COMPARISON OF AGENT INTEROPERABILITY PROTOCOLS (ADAPTED FROM [2], [9])

| Aspect             | MCP (Model Context Protocol)                                 | A2A (Agent-to-Agent)                                         | ACP (Agent Comm. Protocol)                          | ANP (Agent Network Protocol)                              |
|--------------------|--------------------------------------------------------------|--------------------------------------------------------------|-----------------------------------------------------|-----------------------------------------------------------|
| Architecture Model | Client-Server with JSON-RPC primitives                       | Peer-like Client → Remote Agent                              | Brokered Client-Server (Registry + Task Routing)    | Decentralized Peer-to-Peer                                |
| Agent Discovery    | Manual registration or static URL lookup                     | Agent Card retrieval via HTTP                                | Registry-based                                      | Search Engine Discovery                                   |
| Message Format     | JSON-RPC 2.0 with Prompts, Tools, Resources                  | Task + Artifact messaging over JSON                          | Structured multipart messages with MIME-typed parts | JSON-LD with Schema.org and ADP/Meta-Protocol negotiation |
| Transport Layer    | HTTP, Stdio, Server-Sent Events (SSE)                        | HTTP with optional SSE + Push Notifications                  | HTTP with incremental streams                       | HTTP with JSON-LD over TLS                                |
| Target Scope       | Vertical: LLM ↔ Tool/Service                                 | Horizontal: Agent ↔ Agent                                    | Model-Agnostic, Infrastructure-level                | Open Internet agent interconnectivity                     |
| Primary Use Case   | Augment LLMs with external capabilities (e.g., code, search) | Multi-agent workflows within organizational trust boundaries | Secure, typed message exchange for diverse agents   | Cross-platform agent discovery, secure P2P execution      |

3) *Configuration Drift*: Over time, small and unintended configuration changes may accumulate. These changes can move the system away from its original secure setup, creating new security gaps that attackers can exploit [1].

## VI. MCP ECOSYSTEM AND ADOPTION

Although MCP is a promising standard, its long-term success depends on how well it is adopted and maintained in real-world environments. A large-scale measurement study conducted in 2025 by Guo et al. [5] provides useful insights into the current state of the MCP ecosystem.

### A. Market Landscape and Fragmentation

According to the study by Guo et al., a total of 8,401 valid MCP projects were identified, including 8,060 servers and 341 clients, collected from major marketplaces such as MCP.so, MCP Market, and Smithery [5]. The authors describe the ecosystem as “sizable but fragile.” Out of 17,630 raw entries, more than half (50.9%) were found to be invalid, low-quality placeholders, or abandoned prototypes. This indicates that although interest in MCP is growing, many projects are still experimental and not ready for production use [5].

### B. Maintainability and Security Posture

The overall health of the ecosystem also raises concerns. Another study by Hasan et al., which examined 1,899 MCP servers, found that many projects show strong initial development but suffer from maintainability and security problems over time [4]. The key findings include:

- **Maintainability**: About 66% of the servers contained code smells, such as high cognitive complexity, and 14.4% had detectable bugs. These issues suggest that many servers may become difficult to maintain in the long run [4].
- **Security**: Around 7.2% of the servers contained general security vulnerabilities. The most common issue was credential exposure, such as hard-coded API keys, which appeared in 3.6% of the servers. In addition, 5.5% of the

servers were found to be vulnerable to tool poisoning, which is a threat specific to MCP-based systems [4].

- **Supply-Chain Monocultures**: The ecosystem relies heavily on a small number of libraries, such as Spring for Java servers and `pydantic` for Python servers [5]. This creates a monoculture effect, where a vulnerability in a widely used library could impact a large portion of the MCP ecosystem [4], [5].

These results suggest that while MCP successfully addresses interoperability at the protocol level, it also introduces new challenges related to ecosystem security, maintenance, and governance.

## VII. FUTURE RESEARCH AND SCOPE

Future work on the MCP protocol is expected to address the current limitations and expand its capabilities. Several research directions have been identified:

- **Enhanced Security**: One major area of focus is improving data security. This includes stronger encryption methods and new authentication mechanisms specifically designed for MCP-based systems [7]. Researchers are also exploring solutions such as prompt firewalls to detect and block injection attacks [6], along with Zero Trust approaches like Just-in-Time (JIT) access control [8].
- **Centralized Oversight**: Another important direction is the development of a formal package management system and a centralized server registry, similar to platforms like PyPI or NPM. Such a system could help with secure discovery, version control, and prevention of known vulnerable server deployments [1].
- **Improved Interpretability**: Future research also aims to make AI decisions more understandable. Better interpretability tools would help developers and users understand why certain actions were taken, which is important for debugging and building trust in AI systems [7].
- **Multi-Modal Data Support**: Another direction is extending MCP beyond JSON-based data to support multiple data types, such as audio, video, and images. This

would allow MCP to support next-generation multi-modal agents [6].

- **Next-Generation Agents (NGENTs):** Researchers are also exploring the integration of multiple domains, such as text, vision, and robotics, into a single general-purpose agent framework. This would move beyond the current domain-specific models [11].
- **New Application Domains:** MCP concepts can also be applied to other complex, real-time domains, such as adaptive transport systems or building energy management [10], [12].

#### A. Realistic Agent Environments and Evaluation Challenges

Recent studies have shown that deploying autonomous agents in realistic environments is still a difficult problem. The WebArena framework provides a fully functional, self-hosted web environment designed to simulate real-world tasks across domains such as e-commerce, online forums, content management, and collaborative development [24].

WebArena includes a benchmark with more than 800 long-horizon tasks described using high-level natural language instructions. These tasks require agents to perform multi-step reasoning, interact with various tools, and manage changing states across multiple web pages [24].

Experimental results show that even advanced GPT-4-based agents achieve only a 14.41% task success rate, while human participants achieve about 78.24% [24]. This large performance gap highlights the limitations of current agent architectures in complex, real-world environments.

These findings further emphasize the importance of standardized protocols and structured tool interaction frameworks, such as the Model Context Protocol.

#### B. Potential Impact and Applications

The adoption of an MCP-based architecture, as demonstrated in the TripSync case study, has the potential to bring several practical benefits across different domains [6], [7]. Some of the expected impacts include:

- **Reduced Task Complexity:** Both users and developers can benefit from simpler workflows. For example, users can receive complete travel plans without manually searching across multiple platforms [13], [14].
- **Improved User Satisfaction:** Recommendations generated from multiple reliable sources can increase user confidence and overall satisfaction [14].
- **Better Decision-Making:** Context-aware suggestions, such as recommended attractions or optimized travel durations, provide advantages over traditional fragmented systems [13].
- **Operational Efficiency and Scalability:** A decoupled microservice architecture allows individual tools to be scaled or updated independently, making the overall system more robust and maintainable [6], [7].
- **Scalable Data Analytics:** Anonymised interaction data can be used to analyse user behaviour and trends while

maintaining privacy through structured, protocol-based communication [6].

Overall, this architecture provides a foundation for a scalable ecosystem of intelligent and interoperable tools. In the future, it can support integrations with booking platforms, map services, and collaborative travel planning applications [7].

#### VIII. CONCLUSION

Migrating Large Language Models to Deal with Context Using a standardised protocol-oriented architecture, rather than traditional software methods, to build and combine AI applications is an important advance. "TripSync", a trip-planning web application, is used as an example of the use of the Model Context Protocol (MCP) to build modular, flexible, AI-based applications with good context handling. MCP is a client-server framework that provides a standard interface for connecting an LLM to other tools. The multiplexing of contexts onto inputs to the LLM enables multiple applications to use the same LLM with multiple data sources.

At the same time, the introduction of this protocol must be accompanied by a concern about security and ecosystem problems. Although MCP solves many of the interoperability problems, it also creates new security, governance, and long-term maintenance challenges. Research has revealed quality-of-code challenges and vulnerabilities, such as tool poisoning across multiple MCP servers.

By responding to these challenges with enhanced security measures, centralised registries, and community-dictated norms, developers can shape the next wave of AI systems that are more than just powerful; they're also safe and dependable.

#### REFERENCES

- [1] X. Hou, Y. Zhao, S. Wang, and H. Wang, "Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions," *arXiv preprint arXiv:2503.23278v2 [cs.CR]*, 2025.
- [2] A. Ehtesham, A. Singh, S. Kumar, and G. K. Gupta, "A Survey of Agent Interoperability Protocols: Model Context Protocol (MCP), Agent Communication Protocol (ACP), Agent-to-Agent Protocol (A2A), and Agent Network Protocol (ANP)," *arXiv preprint arXiv:2505.02279v1*, 2025.
- [3] N. Krishnan, "Advancing Multi-Agent Systems Through Model Context Protocol: Architecture, Implementation, and Applications," *arXiv preprint arXiv:2504.21030v1*, 2025.
- [4] M. M. Hasan et al., "Model Context Protocol (MCP) at First Glance: Studying the Security and Maintainability of MCP Servers," *arXiv preprint arXiv:2506.13538v4*, 2025.
- [5] H. Guo et al., "A Measurement Study of Model Context Protocol Ecosystem," *arXiv preprint arXiv:2509.25292v2*, 2025.
- [6] P. P. Ray, "A Survey on Model Context Protocol: Architecture, State-of-the-art, Challenges and Future Directions," *TechRxiv*, Apr. 2025.
- [7] P. Pajo, "Smithery.ai: A Model Context Protocol for Enhanced Large Language Model Integration and Cross-Industry Applications," *ResearchGate*, Mar. 2025.
- [8] V. S. Narajala and I. Habler, "Enterprise-Grade Security for the Model Context Protocol (MCP): Frameworks and Mitigation Strategies," *arXiv preprint arXiv:2504.08623v2*, 2025.
- [9] Q. Li and Y. Xie, "From Glue-Code to Protocols: A Critical Analysis of A2A and MCP Integration for Scalable Agent Systems," *arXiv preprint arXiv:2505.03864v1*, 2025.
- [10] H. Li, Y. Xu, and T. Hong, "EnergyPlus-MCP: A model-context-protocol server for ai-driven building energy modeling," *SoftwareX*, vol. 32, p. 102367, 2025.

- [11] Z. Li et al., "NGENT: Next-Generation AI Agents Must Integrate Multi-Domain Abilities to Achieve Artificial General Intelligence," *arXiv preprint arXiv:2504.21433v1*, 2025.
- [12] G. Chhetri et al., "Model Context Protocols in Adaptive Transport Systems: A Survey," *arXiv preprint arXiv:2508.19239v1*, 2025.
- [13] N. Arora, M. R. Shashank, V. Manasa, C. M. Metri, and H. S., "Full Stack Trip Planner Using AI," *International Journal of Research Publication and Reviews*, vol. 6, no. 5, pp. 13444–13449, 2025.
- [14] E. L. González-Sanz, I. Cantador, and A. Bellogín, "LLM-based Generation of Personalized, Context-aware City Tourist Itineraries: A User Study with GPT Trip Planner," in *Proc. Workshop on Recommenders in Tourism (RecTour 2025)*, 2025.
- [15] Anthropic, "Introducing the Model Context Protocol," *Anthropic News*, Nov. 2024. [Online]. Available: <https://www.anthropic.com/news/model-context-protocol>
- [16] L. Wang et al., "A survey on large language model based autonomous agents," *Frontiers of Computer Science*, vol. 18, no. 6, 2024.
- [17] Z. Xi et al., "The rise and potential of large language model based agents: A survey," *arXiv preprint arXiv:2309.07864*, 2023.
- [18] OpenAI, "Function calling in OpenAI models," *OpenAI Documentation*, 2023. [Online]. Available: <https://platform.openai.com/docs/guides/function-calling>
- [19] T. Schick et al., "Toolformer: Language models can teach themselves to use tools," in *NeurIPS*, 2023.
- [20] S. Yao et al., "ReAct: Synergizing reasoning and acting in language models," in *ICLR*, 2023.
- [21] Google, "Agent2agent (a2a) protocol documentation," 2024. [Online]. Available: <https://google.github.io/A2A/>
- [22] IBM BeeAI, "Introduction to agent communication protocol (acp)," 2024. [Online]. Available: <https://docs.beeai.dev/acp/alpha/introduction>
- [23] Agent Network Protocol Contributors, "Agent network protocol (anp)," 2024. [Online]. Available: <https://github.com/agent-network-protocol/AgentNetworkProtocol>
- [24] S. Zhou et al., "WebArena: A Realistic Web Environment for Building Autonomous Agents," in *Proc. ICLR*, 2024.