Unified MCP Framework for Context-Aware AI Agents in Software Development

A major project report submitted in partial fulfilment of the requirement for the award of degree of

Bachelor of Technology

in

Computer Science & Engineering

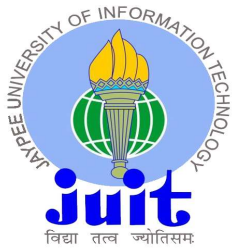
Submitted by

Harsh Kumar, Anjali Panwar, Om Shree, Naviya Tickoo

(221031040, 221030113, 221030410, 221030050)

Under the guidance & supervision of

Dr. Aman Sharma (Assistant Professor)



Department of Computer Science & Engineering and Information Technology

Jaypee University of Information Technology, Waknaghat, Solan - 173234 (India)

December 2025

Candidate's Declaration

We hereby declare that the work presented in this major project report entitled **'Unified MCP Framework for Context-Aware AI Agents in Software Development'**, submitted in partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science & Engineering**, in the Department of Computer Science & Engineering and Information Technology, Jaypee University of Information Technology, Waknaghat, is an authentic record of our own work carried out during the period from July 2025 to December 2025 under the supervision of Dr. Aman Sharma.

We further declare that the matter embodied in this report has not been submitted for the award of any other degree or diploma at any other university or institution.

Name: Harsh Kumar
Roll No.: 221031040
Date: 01/12/2025

Name: Anjali Panwar
Roll No.: 221030113
Date: 01/12/2025

Name: Om Shree
Roll No.: 221030410
Date: 01/12/2025

Name: Naviya Tickoo
Roll No.: 221030050
Date: 01/12/2025

This is to certify that the above statement made by the candidates is true to the best of my knowledge.

Supervisor Name: Dr Aman Sharma
Designation: Assistant Professor (Senior Grade)
Department: Department of CSE & IT

Date: 01/12/2025
Place: JUIT, Solan

Supervisor's Certificate

This is to certify that the major project report entitled **'Unified MCP Framework for Context-Aware AI Agents in Software Development'**, submitted in partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science & Engineering**, in the Department of Computer Science & Engineering and Information Technology, Jaypee University of Information Technology, Waknaghat, is a bona fide project work carried out under my supervision during the period from July 2025 to December 2025.

I have personally supervised the research work and confirm that it meets the standards required for submission. The project work has been conducted in accordance with ethical guidelines, and the matter embodied in the report has not been submitted elsewhere for the award of any other degree or diploma.

Supervisor Name: Dr Aman Sharma
Designation: Assistant Professor (Senior Grade)
Department: Department of CSE & IT

Date: 01/12/2025
Place: JUIT, Solan

ACKNOWLEDGEMENT

I express my sincere gratitude to the Almighty for granting me the strength, perseverance, and clarity of mind needed to complete this project successfully. I am deeply indebted to my supervisor, **Dr. Aman Sharma**, Assistant Professor, Department of Computer Science & Engineering and Information Technology, Jaypee University of Information Technology, Waknaghat, for his continuous guidance and encouragement throughout the course of this work. His expertise in the field of **Artificial Intelligence**, along with his insightful feedback, patient supervision, and constructive suggestions at every stage, has been invaluable. This project would not have taken its present shape without his consistent motivation and academic support. I would also like to extend my heartfelt thanks to all the faculty members and staff of the Department of CSE & IT for their kind cooperation, timely assistance, and encouragement during the development of this project. Their contributions, whether direct or indirect, have been instrumental in helping me progress smoothly.

Finally, I am grateful to my peers, friends, and family members for their constant support, understanding, and encouragement throughout this journey.

Harsh Kumar, 221031040
Btech CSE (4^{th} year)

Anjali Panwar, 221030113
Btech CSE(4^{th} year)

Om Shree, 221030410
Btech CSE(4^{th} year)

Naviya Tickoo, 221030050
Btech CSE(4^{th} year)

TABLE OF CONTENTS

CONTENT **PAGE NO.**

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS, SYMBOLS OR NOMENCLATURE

S. No.	Abbreviation / Symbol	Full Form / Meaning
1	AI	Artificial Intelligence
2	LLM	Large Language Model
3	MCP	Model Context Protocol
4	MCIP	Model Contextual Integrity Protocol
5	API	Application Programming Interface
6	IDE	Integrated Development Environment
7	UI	User Interface
8	CSS	Cascading Style Sheets
9	npm	Node Package Manager
10	VS Code	Visual Studio Code
11	JSON	JavaScript Object Notation
12	URL	Uniform Resource Locator

ABSTRACT

Modern software development often requires developers to switch between multiple environments such as desktop applications, browsers, and GitHub. This fragmented process creates inefficiency, integration challenges, and slower workflows. To overcome these issues, this project proposes a Unified Model Context Protocol (MCP) Framework that integrates Desktop MCP, Browser MCP, and GitHub MCP through a central Host MCP Server, offering a unified communication layer for seamless interaction.

The system is designed to provide a connected environment where large language models (LLMs) and AI agents assist developers in coding, debugging, and collaboration. A full-stack platform with modern frontend, backend, and database support is being developed to deliver an intuitive and interactive interface. Its modular architecture ensures scalability, flexibility, and adaptability for future extensions. Additional features such as containerization, cloud deployment, and database management further strengthen reliability and performance.

By enabling smooth communication across MCP servers, the framework allows developers to interact with LLM-powered agents in a more efficient and context-aware manner. This reduces manual context switching, improves productivity, and streamlines workflows. In the long run, the Unified MCP Framework aims to transform software engineering practices by promoting intelligent, agent-driven development tools that simplify collaboration, accelerate problem-solving, and support the growing need for scalable AI-assisted solutions.

CHAPTER 1: INTRODUCTION

1.1 INTRODUCTION

The field of software development has changed at an exceptional pace over the last decade. Many tools now support a developer to work faster and with fewer errors. Code completion, syntax checking, debugging assistance and automated testing are all supported by intelligent systems. It is mainly due to the increased use of artificial intelligence in development environments. Most of the modern tools have features that offer code understanding, suggestions and automation of many tasks that used to be repetitive and time-consuming.

Although these improvements include productivity gains, they don't solve a common difficulty faced by almost each developer. That is, today software development requires working across several different environments. A developer could write code in a local IDE, search for documentation on the web, review commits on GitHub, check logs in a browser console, or even view deployment status on a cloud platform[1]. These continuous changes prevent concentration, break continuity and quite often lead to lost context. Once the developer switches over to a browser or another tool, even advanced assistants available in the IDE cannot help. So the result is scattered flow with less efficiency.

At the same time, the abilities of modern AI systems evolved from simply suggesting code to understanding the general intent of the task at hand. They read files, analyze logs, generate documentation, refactor code, review pull requests, debug and do multi-step tasks in a coordinated manner[2][3]. And because of that, many organizations treat intelligent agents as the next step forward in improving software engineering practice. Such agents would be capable of supporting developers continuously via an entire workflow instead of only within a single tool.

There is a major limitation, though, preventing these agents from being truly effective. Most tools still operate in isolation from one another. An assistant inside of an IDE cannot automatically help once the task moves over to the browser or to GitHub. Another assistant built into the browser cannot access the developer's local files or project structure. But without a means of sharing context across these environments, even the most powerful agents are confined to a small part of the developer’s workflow [4].

That could finally change in 2024, with the introduction of the Model Context Protocol, or MCP. MCP represents a promising answer to these problems. MCP offers a structured way for intelligent systems to communicate with external tools and services. It provides a simple, secure method by which an agent can discover what tools are available, what actions those tools can perform and interact with them in predictable

ways [1][5]. MCP has been designed to help agents maintain this context even when tasks shift between files and repositories or from web searches to code analysis. Following the principles of MCP, tools remain transparent, safe and easy to manage [4].

This project will make use of MCP to create a Unified MCP Framework that brings together into one cohesive space three of the most important environments used by developers in their day-to-day work: the local filesystem, the web browser and GitHub. For this purpose, the system contains the respective MCP servers for each environment. A central Host MCP Server coordinates these and ensures that the agent is allowed to switch from one environment to another without losing context. An example would be reading a local file, searching the related documentation online and inspecting a repository on GitHub, all inside one continuous workflow [6].

Such a system becomes clearer when you look at the complexity of modern development work. One feature might require a person to read the documentation, check the code history, edit files, look for examples, test output and push changes to a repository. The result of trying to manage all this by hand is repetition and loss of focus. A unified, context-aware framework has potential to reduce this fragmentation since it can help the agent make sense of how these tasks relate to each other. With shared context and structured tool access, the overall process is more organized and efficient [3].

Another important aspect is that of safety. Intelligent agents need clear boundaries so that their actions remain transparent and under control. MCP tells us exactly how we can use these tools and what parameters they take. That makes it a lot easier to keep the track of what's going on[7]. A system built on these principles allows developers to benefit from automation while still maintaining control of their environment[8]. However, MCP-related studies show significant security risks such as tool poisoning, malicious servers and ecosystem-wide vulnerabilities [8][9]. Safety frameworks like MCIP attempt to mitigate these risks through guard models and controlled tool invocation [9].

However, security studies show that MCP introduces new attack surfaces such as tool poisoning, puppet attacks and malicious external resources. Recent work also highlights ecosystem-wide vulnerabilities and poor maintainability in public MCP servers. In response, safety frameworks such as MCIP provide guard models and structured defences that improve safe tool-invocation behaviour [8], while enterprise-grade guidelines propose zero-trust designs for real-world deployments [9].

The importance of unified, agent-assisted workflows is growing rapidly in industry. Tools like GitHub Copilot have already shown how much difference intelligent assistance can make. However, they remain limited to the tools where they are installed. A framework which allows for smooth movement between environments represents a natural next step in the evolution of development tools. It allows developers to focus more on design and problem-solving rather than tool management and platform switching[2]. The Unified MCP Framework developed in this project is motivated by the need to reduce workflow interruptions, support continuous assistance across different platforms and bring structure to agent-based automation. It also ties together our desktop, browser and GitHub under one coordinated system, enhancing productivity, clarity and overall experience in modern software development [6][7].

1.2 PROBLEM STATEMENT

Developers lose productivity when switching between IDEs, browsers, and GitHub, and current AI assistants cannot coordinate across these tools securely. Increasing risks like unauthorized tool use demand a safer, unified workflow. This project proposes a Unified MCP Framework that securely integrates Desktop, Browser, and GitHub MCP servers to create a seamless, context-aware developer environment.

1.3 OBJECTIVES

• Design a flexible framework that integrates multiple MCPs (desktop, browser, GitHub) so developers and automated processes can communicate through a unified protocol.

• Build a central host to coordinate and manage all communication between the three MCPs and their connected agents.

• Develop a full intelligent coding environment—front-end, back-end, and database—enabling developers to collaborate with AI agents for coding, debugging, and end-to-end workflow management.

1.4 SIGNIFICANCE AND MOTIVATION OF THE PROJECT WORK

Our project is highly relevant because it focuses on an important gap in today's AI-assisted development workflows. Although there is a growing presence of assistants powered by LLMs, they normally only work within a single environment, such as within an integrated development environment or a code editor. Once the developer moves to another platform, the assistant cannot keep up its context or support the workflow any further. Such fragmentation limits efficiency and prevents these tools from matching the way modern developers work across multiple platforms.

Simultaneously, AI agents have begun to emerge as one of the fastest-growing areas of technology. Contrary to simple code assistants, agents can operate independently, manage multi-step tasks and even optimize workflows. For example, in 2024, the value of the global AI Agents market was an estimated USD 5.40 billion and is expected to reach USD 50.31 billion by 2030, according to the report by Grand View Research dated 2024[10]. This rapid growth indicates their rising role in improving productivity and collaboration in software engineering.

Some platforms have already started pushing in this direction. For example, Figma has introduced support for MCP servers that let the AI agents work with both design and code, rather than being kept in isolation. Indeed, all these show the industry is moving to more connected and agent-friendly systems [16]. The motivation for this work is the necessity to minimize workflow interruptions and build tools that make full use of agent-driven development. Such a unified MCP framework will improve developer productivity by allowing seamless collaboration and providing a scalable foundation for the next generation of AI-driven tools.

The project has additional academic value. It illustrates how protocol-driven agent architectures can be used in realistic development contexts. It closes the gap between research on AI agents and their actual industrial application in software engineering processes. The framework integrates tool servers, orchestration logic and a user-facing interface into an overall model that is extensible for future research or industrial applications. On the learning front, this provides hands-on experience with modern technologies like FastAPI, Playwright, React, PyGithub and LLM-based intent analysis. It underlines the need for safe tool execution, sandboxing and API governance that are increasingly relevant as AI agents become more powerful and widespread [12]. With this in place, the experience gained from building the unified framework is of great value both from an academic point of view but also from that of a real-world engineering role where AI-assisted tooling is the new standard.

1.5 ORGANIZATION OF PROJECT REPORT

This project report walks us through every stage of building the Unified MCP Framework. Each chapter builds upon the previous one, ensuring a clear and detailed understanding of the motivation, design decisions, implementation methods and evaluation of the system . The structure of the report is as follows:

Chapter 1: Introduction

This chapter presents the basis of the project by describing the evolution of AI-assisted development and the challenges of context switching among multiple environments. It explains the motivation for creating a unified agent framework , outlines the problem statement, lists the objectives and discusses the significance of the work. This chapter establish the need for an MCP-based solution that integrates filesystem operations , web research and GitHub interactions under a single coherent system.

Chapter 2: Literature Survey

This chapter reviews existing research , tools and technologies related to AI Agents, tool-calling architectures , browser automation , repository inspection and developer productivity workflows. It includes studies on early LLM-based coding assistants, recent developments in agent frameworks and the introduction of the Model Context Protocol. By examining related work, the chapter highlights the gaps in current systems, especially the lack of unified cross-platform context sharing, which forms the core motivation for this project.

Chapter 3: System Development

This chapter explains the complete development of the Unified MCP Framework. It begins with a detailed requirements and analysis section, covering both hardware and software needs essential for running multiple MCP servers and automation workflows. The chapter then introduces the system architecture, describing how the Host MCP Server coordinates , communicate with the Filesystem , Browser and GitHub MCP servers. It discusses the design strategies, component-level behaviour and the reasoning behind key architectural decisions , such as sandbox isolation , schema-based tool invocation and asynchronous execution. The implementation section presents important code segments, tool definitions and integrations with Playwright, PyGithub and the Gemini API. The chapter concludes with the challenges encountered during development and how they were addressed.

Chapter 4: Testing

This chapter describes how we checked the framework's accuracy , stability and the reliability of the system . It includes functional tests for each MCP tool, integration tests to ensure if the servers work well together and performance checks for both browser automation and repository access. We get actual test cases , expected outputs and actual outcomes are documented to demonstrate the robustness of the system. The chapter also assesses how well the framework handles invalid inputs , path restrictions, network variability and tool execution errors.

Chapter 5: Results and Evaluation

This chapter is all about the results and evaluation . It presents the outcomes of the developed framework and summarizes its performance across various use cases such as file manipulation, structured web extraction and repository inspection. Results are highlighted to show the accuracy of tool execution, responsiveness of the orchestration engine and reliability behaviour. When appropriate, comparisons are drawn with existing tools and conventional workflows in order to demonstrate improvements in efficiency, transparency and usability. The limitations of real-world testing are discussed .

Chapter 6: Conclusions and Future Scope

This chapter concludes the report with a summary of the contributions of the project and its impact on AI-assisted development. The effectiveness of the MCP-based architecture is reflected upon and it is discussed how the unified framework solves problems faced by developers due to environment switching. Potential directions for future work are also identified, including support for more MCP servers, enhanced agent capabilities, options for cloud deployment and integration with more developer platforms.

After the main chapters, the report is complete with a References section formatted in IEEE style, documenting all research papers, tools and online sources referred to throughout the project. The Appendix includes supplementary material , comprising code snippets, screenshots , architectural diagrams, tool schemas, sample outputs and a user guide to run the system. A duly signed Plagiarism Certificate is attached at the end to ensure academic integrity.

CHAPTER 2: LITERATURE SURVEY

2.1 OVERVIEW OF RELEVANT LITERATURE

Recently, the Model Context Protocol (MCP) marks an essential standard towards allowing large language models (LLMs) and autonomous agents to communicate with external software and other tools, as well as with sources and computational systems of real-world data. Newer literature demonstrates that MCP is transitioning out of an experimental concept and into an ecosystem that has peer-reviewed academic literature, empirical measurements and application-specific to a domain [13][14].

One of the key trend in the MCP literature is the onset of real-world, proven applications in the field of engineering. Avila et al. (2025) combine GPT-based models with the OpenSeesPy structural analysis engine via MCP and show that the workflows of MCP are equally accurate as the traditional engineering tools. Their paper indicates that direct LLM reasoning is associated with large numerical errors, whereas MCP guarantees deterministic calls to solvers and sound results. In a similar way, Li et al. (2025) introduce an EnergyPlus-MCP server that makes available to LLMs an entire suite of building-energy simulation applications. Their results indicate that MCP maintains correctness in simulation as well as minimizing the human effort to develop the building-energy modelling [27].
These two peer-reviewed articles are good evidence that MCP is able to facilitate high level of scientific and engineering activities.

MCP research is also based on the architectural and standardisation-oriented studies. Karimova (2025) examines the architecture of the protocol and points out its vendor-neutral architecture, typed tool schemas and modular resource architecture [13]. MCP Experience complementary system demonstrations that were published via Authorea illustrate how MCP can integrate between LLMs and other external systems via a single JSON-RPC interface [16]. Such works explain that MCP has advantages in terms of minimizing integration overhead and reusable software components.

Guideline Guo et al. (2025) have measured public MCP registries comprehensively and it is the first study to examine the ecosystem-level behaviour of MCP. They observed that many servers are duplicated, abandoned or lowly designed by examining over seventeen thousand entries. In their work, they find dependency monocultures and irregular maintenance between repositories, as well as issues of trust in supply-chain and reliability of servers [14]. The work offers the initial empirical foundation in the context of comprehending the MCP adoption patterns and the infrastructure health in the real-world setting.

Another significant research direction is security and safety. According to Song et al. (2024), MCP-specific attacks have four new classes. Their proof-of-concept experiments show that the behaviour of models can be manipulated by compromised or deceptive tools demonstrating dangers in the form of tool poisoning, puppet attacks and malicious external resource [11]. Similar results are demonstrated by Croce and South (2024), who indicate that by abusing cross-tool interactions, even a fairly basic malicious MCP server can steal valuable financial data [13]. Hasan et al. (2024) adopt a wider perspective by scanning close to two thousand MCP servers and reveal that most of them have security vulnerabilities or implementation problems that can result in exposing users to risk [14].

In reaction to these issues, a number of works suggest safety-enhancing extensions and defensive models. Narajala and Habler (2024) introduce a business-oriented security system blueprint of MCP implementations that focuses on Zero-Trust concepts and defence-in-depth approaches [15]. However, the most detailed safety plan is the Model Contextual Integrity Protocol (MCIP) that Jing et al. (2025) propose. Their work shows weaknesses in the safety controls of MCP, introduces a rich taxonomy of unsafe behaviours and introduces MCIP-bench, a benchmark data set of assessing the safety of the LLM when using tools. They also generate a safety-sensitive guard model, which highly enhances the safe tool-invocation rates in various LLMs [18].

Multi-agent settings have also been examined using MCP. Krishnan (2024) introduces a system that applies MCP to ensure that context is consistent between the agents that cooperate to achieve better coordinated problem solving [16]. The article by Avram et al. (2024) is a multi-agent pipeline of disinformation detection designed by MCP and achieves a high classification score, which proves the utility of MCP in distributed intelligent systems [20].

Further general ecosystem and protocol analyses help to better explain the role of MCP. The architectural structure and lifecycle of MCP are mapped and the main security threats are identified by Hou et al. (2024) and the research perspective of the future is outlined [22]. An interoperability survey done by a wider group of Ehtesham et al. (2025) compares MCP to other new agent protocols. They find that MCP is now the most viable choice to make accuracy, typed tool invocation and that complementary protocols like ACP, A2A and ANP are used to carry out communication, delegate tasks and decentralised agent discovery [19].

Bringing these findings into one, the literature suggests four key trends. First, MCP offers a stable interface between LLMs and deterministic tools and scientific solvers to enhance accuracy and reproducibility. Second, even as the ecosystem grows at a high rate, it is inconsistent in quality as evidenced in large scale measurements studies [14]. Third, MCP brings about security risks which need effective governance and even though there are some early safety structures, extensive implementation of defences are not proven yet[11]. Lastly, the existing studies mostly assess MCP in single settings. There are very limited studies on how MCP can be used to enable

context continuity between different developer tools including IDEs, browsers or code repositories[13]. This gap is very applicable to the actual software-engineering processes. The current project is based on the insights. It suggests an integrated, cross-environment MCP which maintains developer context across the various tools, integrates safety concepts in MCIP but also meets ecosystem-wide issues that have been identified in recent measurement research.

Table 2.1: Overview of Relevant Literature on Model Context Protocol (MCP)

S.No.	Author& Paper Title	Journal/Conference (Year)	Tools/Techniques/Dataset	Key Findings/Results	Limitations/Gaps Identified
1	**Carlos Avila et al. Human–AI Teaming in Structural Analysis with MCP[26]**	*MDPI* (2025)	GPT-MCP, OpenSeesPy	MCP-driven structural analysis matches solver accuracy; eliminates free-text LLM errors	Domain-specific; strict schema required
2	**Han Li et al. EnergyPlus-MCP: MCP Server for Building Energy Modelling[27]**	*SoftwareX ScienceDirect* (2025)	EnergyPlus-MCP; 35 tools	Enables accurate AI-driven building-energy simulations; reduces manual work	No production-scale evaluation
3	**Sevinj Karimova - Model Context Protocol: A Standardisation Analysis[23]**	*UNEC JCS&DT* (2025)	Architectural analysis	Discusses MCP architecture, primitives and adoption challenges	Lacks empirical data
4	**Vallikranth Ayyagari - MCP for Agentic AI:**	*IJCESEN* (2025)	MCP primitives; architecture	Argues MCP's vendor-agnostic design	Needs scalability & governance studies

	Contextual Interoperability [21]			enhances interoperability	
5	**Huihao Jing et al. - MCIP: Protecting MCP Safety via Model Contextual Integrity Protocol[18]**	*EMNLP* (2025)	MCIP-bench; taxonomy; guard model	Defines unsafe behaviours; improves LLM safety during tool use	Needs real-world adoption testing
6	**Authorea - MCP: Bridging AI and External Systems[25]**	*Authorea* (2025)	MCP integration demos	Shows general MCP integration patterns across systems	Preprint; lacks quantitative evaluation
7	**Alexandru-Andrei Avram et al. - MCP for Disinformation Detection[20]**	Preprint (2024)	Multi-agent pipeline	Achieves 95.3% accuracy and F1 0.964 using MCP coordination	External API dependency; lacks adversarial testing
8	**Naveen Krishnan -Advancing Multi-Agent Systems Through MCP[16]**	Preprint (2024)	Multi-agent MCP architecture	Shows improved coordination and context retention	No large-scale benchmarking
9	**Jun Cui -MCP Capability in AIOps R&D Processes[17]**	Preprint (2024)	AIOps tests: 1200 req/s	Demonstrates high-throughput, low-latency MCP integration	Single lab environment
10	**Hao Song et al. - Beyond the Protocol: Attack**	*arXiv* (2024)	PoC attacks on 5 LLMs	Identifies Tool Poisoning, Puppet, Rug Pull,	No mitigation or auditing framework

	Vectors in MCP[11]			Malicious Resources	
11	**Shiqing Fan et al. MCPToolBench ++Benchmark[12]**	*arXiv* (2024)	1.5K Q&A; 4K+ servers	Large-scale benchmark for tool invocation evaluation	Limited tool diversity coverage
12	**Nicola Croce & Tobin South - Trivial Trojans[13]**	*arXiv* (2024)	Malicious MCP weather server	Shows minimal malicious servers can exfiltrate sensitive data	PoC only; prevalence unknown
13	**Mohammed Mehedi Hasan et al. - MCP at First Glance[14]**	*arXiv* (2024)	Scan of 1,899 servers	Finds 7.2% generic vulnerabilities and 5.5% MCP-specific tool poisoning	Needs MCP-specific detection tools
14	**Vineeth Sai Narajala & Idan Habler - Enterprise-Grade MCP Security[15]**	*arXiv* (2024)	MAESTRO; Zero Trust	Provides actionable security mitigations	Not validated in real deployments
15	**Abul Ehtesham et al. - Survey of MCP, ACP, A2A, ANP[19]**	*arXiv* (2025)	Protocol comparison	MCP best for tool invocation; others complement it	No experimental benchmarks
16	**Xinyi Hou et al. - MCP Landscape, Threats, and Future Directions[22]**	*arXiv* (2024)	Architecture & threat taxonomy	Maps risks: name collision, spoofing, authentication gaps	No implemented defences

17	**Hechuan Guo et al. - Measurement Study of MCP Ecosystem[24]**	*arXiv* (2025)	MCPCrawler; 17.6K entries	Finds duplication, poor maintenance, dependency monocultures	Snapshot data; no runtime analysis

2.2 KEY GAPS IN THE LITERATURE

1. Ecosystem Quality and Reliability

Large-scale studies by Guo et al. (2025) and Hasan et al. (2024) show that a significant proportion of MCP servers are incomplete, outdated or poorly maintained [14][24]. Many entries in public registries are duplicated or lack meaningful functionality. This affects trust, reliability and long-term sustainability. There is currently no official mechanism for verifying server quality, certifying capabilities or enforcing lifecycle management.

2. Securıty Vulnerabılities and Limited Defensive Coverage

MCP introduces new security challenges because models can trigger external actions. Attack studies (Song et al., 2024; Croce & South, 2024) demonstrate that compromised or deceptive servers can mislead LLMs or exfiltrate user data [11][13]. Although frameworks such as MCIP (Jing et al., 2025) and the enterprise security guidelines of Narajala and Habler (2024) offer promising defences [18][15], these solutions have not yet been adopted widely or tested across large, diverse deployments. Real-world safety under adversarial conditions remains largely untested.

3. Lack of Cross-Environment and Cross-Tool Continuity

Most MCP research focuses on a single tool, a single domain or a controlled environment. No existing study proposes a unified MCP architecture that keeps context consistent across IDEs, browsers, terminals and repository platforms [21][23]. This is a major gap, especially since developers routinely move across multiple environments during daily workflows.

4. Limited Benchmarks Covering Realistic Tool Diversity

MCPToolBench++ expands the evaluation landscape, but its coverage is still significantly smaller than the fast-growing MCP ecosystem [20]. Many real-world tool types, multi-step pipelines and agents requiring rich non-text outputs are not fully represented. This limits the ability to compare LLMs and to stress-test tool-use reliability.

5. Minimal Long-Term or Production-Scale Evaluations

While some studies report high throughput or solver accuracy, there is little research on how MCP behaves in production settings over long durations. Issues such as performance degradation, permission fatigue, runtime failures, update conflicts and data-governance challenges are not yet well understood [18].

6. Mıssing Standards for Server Discovery, Permissions and Governance

Works by Hou et al. (2024) and Ayyagari (2025) highlight gaps in naming conventions, server discovery, permission interfaces and installation security [21][22]. Without governance standards, users are at risk of installing spoofed or unsafe servers and developers lack clear guidelines for secure MCP server design. These gaps motivate the present project. By developing a unified multi-environment MCP framework, integrating safety principles from MCIP and incorporating ecosystem insights from measurement studies, our system aims to provide a secure and reliable foundation for real-world developer assistance.

CHAPTER 3: SYSTEM DEVELOPMENT

3.1 REQUIREMENTS AND ANALYSIS

The Unıfıed MCP Framework addresses one of the key poınts developers face today : constant context swıtchıng between tools and envıronments. Software engıneers have to work sıde-by-sıde ın local IDEs, browsers, GıtHub reposıtorıes, documentatıon and cloud dashboards. The fragmentatıon causes broken contınuıty and decreases productıvıty even when usıng AI-powered tools. Desıgnıng a system that can orchestrate Desktop MCP, Browser MCP and GıtHub MCP usıng one sıngle Host MCP Server requıres statıng hardware and software requırements able to provıde real-time AI-assisted development, browser automation and secure tool execution.

3.1.1 Hardware Requirements

Hardware is the foundation requıred to run several backend servıces, browser automatıon engınes and a full-stack development envıronment together. Whıle the Unıfıed MCP Framework does not need specıalızed hardware such as GPUs , ıt does requıre a stable and moderately powerful machıne to support FastAPI servers , React development servers , Playwrıght's Chromıum engıne and GıtHub API operatıons effıcıently.

Processor Requirements (CPU)

A multi-core CPU is required to run multiple tasks asynchronously, such as tool routing, API calls and browser automation. Suitable options include:

- Intel Core i5 / i7 (10th generation or higher)
- AMD Ryzen 5 / Ryzen 7 series

  These handle multiple asynchronous FastAPI endpoints along with Playwright browser processes without performance degradation. In case of heavier simultaneous automation tasks, high-end processors such as Intel Core i9 or Ryzen 9 offer additional stability.

Memory Requirements (RAM)

Browser-based automation and running multiple development servers can easily consume memory.

- **Minimum:** 8 GB RAM
- **Recommended:** 16 GB RAM or above

  This ensures smooth tab handling in Playwright, fast project reloads in React and stable execution of parallel MCP servers.

Storage Requirements (SSD/HDD)

Storage speed is very important for tool execution, sandbox operations, dependency installation and repository interactions.

- **SSD (256 GB minimum) :** for fast access to sandbox files, browser cache , npm installations and Python virtual environments.
- **HDD (500 GB – 1 TB optional) :** for storing logs, cloned repositories, and testing artifacts.

3.1.2 Software Requirements

The software requirements of the Unified MCP Framework define the overall ecosystem that is required to develop, test and deploy all components of the system. Since this project integrates several asynchronous servers, browser automation tools , a React-based user interface and authenticated interactions with external APIs , the software environment has to be carefully structured so that each portion of the framework communicates reliably with the others. The aim is to establish a development environment that is stable , secure and yet flexible enough to support complex tool orchestration without imposing undue overheads on the system.

A. Operating System Environment

The framework has been developed with the aim of working across the most commonly used desktop operating systems so that developers can be supported in various environments. It currently supports Windows 10 and 11, major distributions of Linux like Ubuntu, and macOS systems. All three are usable; however, the overall best experience is achieved using Linux because of its predictable dependency management and great integration with Python-based backend frameworks.

Windows also works well but may require additional configuration for Playwright and related browser dependencies . MacOS users can run the framework effectively, though sometimes Playwright's installation requires platform-specific adjustments. Whatever the choice of operating system, the environment needs to support the stable running of FastAPI servers, browser automation layers and JavaScript build tools-these form the functional backbone of the project.

B. Backend Frameworks and Libraries

The Unified MCP Framework's backend is implemented in Python because it is a dynamic general-purpose language with a rich ecosystem of libraries, especially for supporting asynchrony. Python allows the implementation of different servers for the MCP, still communicating seamlessly via a central orchestrator.
Each backend component is based on a set of selected libraries for performance, clarity and reliability.

- FastAPI: main web framework for the Host MCP Server ; provides automatic schema generation, asynchronous request handling and high performance.
- Playwright: enables scripted navigation, searching and scraping of webpage content.

- PyGithub: allows integration with GitHub repositories , simplifying directory structure access, metadata retrieval and file content handling.

Standard Python modules like pathlib (for secure path manipulation), shutil and tempfile (for atomic file operations) and asyncio (for asynchronous task scheduling) are also used.

A dedıcated Python vırtual envıronment ıs used to ısolate these dependencıes , avoıdıng versıon conflıcts and makıng deployment sımpler. Thıs setup allows developers to reproduce the same envıronment acıross systems and test updates on ındıvıdual MCP servers wıthout affectıng the entıre system.

C. Frontend Technologıes

The frontend of the Unıfıed MCP Framework manages communıcatıon between the user and backend servers. It ıs built usıng React 19 , provıdıng modularıty , quıck re-renderıng and real-tıme updates. The buıld system uses Vıte, ensuring instant hot reloading and fast build times.

Custom CSS and lightweıght ıcon lıbrarıes maıntaın clarıty and accessıbılıty. The Fetch API handles communıcatıon between the frontend and backend, usıng JSON-formatted requests and structured FastAPI responses.

The ınterface ıs responsıve, adaptıng to dıfferent screen sızes and devıces. Transparency ıs emphasızed where every tool call shows ıts arguments and raw outputs, helpıng users understand results. Thıs traceabılıty dıstınguıshes ıt from other agent frameworks that hıde backend actıvity behind opaque APIs.

D. Browser Automation Dependencies

Browser automation is central to the Browser MCP Server , which performs search queries, parses results, visits websites and extracts content. These are handled through Playwright , requiring Chromium during setup.

Chromium is preferred for its headless execution , allowing automation in the background. While powerful , browser automation faces challenges such as network delays , dynamic pages and OS compatibility. The project mitigates these by including necessary binaries and maintaining consistent versions across machines.

E. Token and Environment Configuration

Secure communication between the servers and APIs depends on an .env configuration file in the backend directory . This stores sensitive information such as:

- Gemini API Key : for summarization and intent analysis within the Host MCP Server.
- GitHub Personal Access Token : for authenticated interactions with private repositories.

Storing credentials as environment variables keeps them secure and easily replaceable if compromised. This setup supports flexible deployment where different credentials can be used for development , testing and production without making changes to the codebase.

F. Developer Tools and Supporting Software

Modern development tools are required for debugging , testing and version control:

- Node.js and npm : build and run the frontend.
- Python 3.10+ for backend execution and async support.
- Git for version control and collaboration.
- VS Code or PyCharm for managing code structure, syntax highlighting and debugging.

The software stack balances performance, usability and extensibility - Python powers asynchronous servers and integrations ; FastAPI provides high-performance routing ; Playwright delivers automation; PyGithub handles repositories ; React and Vite ensure an efficient, interactive UI and environment-based configuration keeps the system secure and flexible.

3.2 PROJECT DESIGN AND ARCHITECTURE

3.2.1 Architectural Overview

The architecture follows a multi-layered pattern to maintain clarity and isolate concerns. Each layer performs a distinct role in orchestrating tool execution , processing user queries and presenting outputs back to the user.

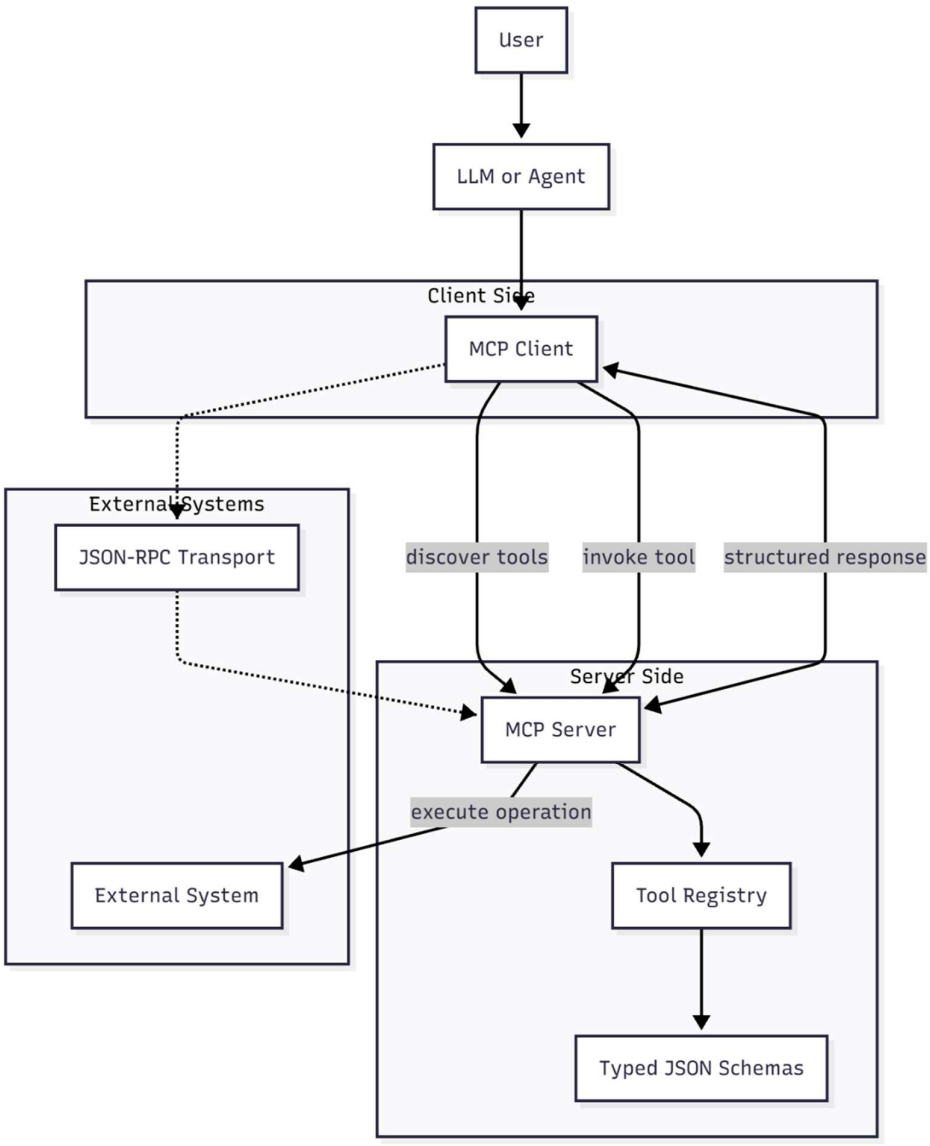


Figure 3.1: Conceptual Architecture of the Model Context Protocol (MCP)

1. Client Layer (React Frontend)

This layer provides the user interface where developers interact with the system. It collects the user queries , sends them to the backend and displays tool responses , raw execution logs and summaries. Its real-time state management and clean layout make it easy to follow complex agent behaviours.

2. Host Orchestration Layer (FastAPI Host MCP Server)

This is the core intelligence layer . It -:

- Loads all tool servers
- Interprets user instructions
- Selects the correct tool server
- Executes tool calls
- Merges tool output with LLM summaries
- Returns structured results to frontend

This layer ensures seamless coordination between all servers.

3. MCP Tool Server Layer

This layer contains specialized servers:

- **Filesystem MCP Server**
- **Browser MCP Server**
- **GitHub MCP Server**

Each server exposes tools following strict schemas to ensure safe , predictable interactions.

4. External Integration Layer

This layer includes external dependencies:

- Playwright (browser automation)
- Chromium browser
- GitHub API
- Google Gemini for summarization

These services extend the system’s capabilities beyond local operations.

5. Storage & Sandbox Layer

The filesystem tools operate in the secure mcp_sandbox directory. This layer ensures:

- Isolation from host filesystem
- Prevention of path traversal
- Controlled read / write limits

3.2.2 Component-Level Design

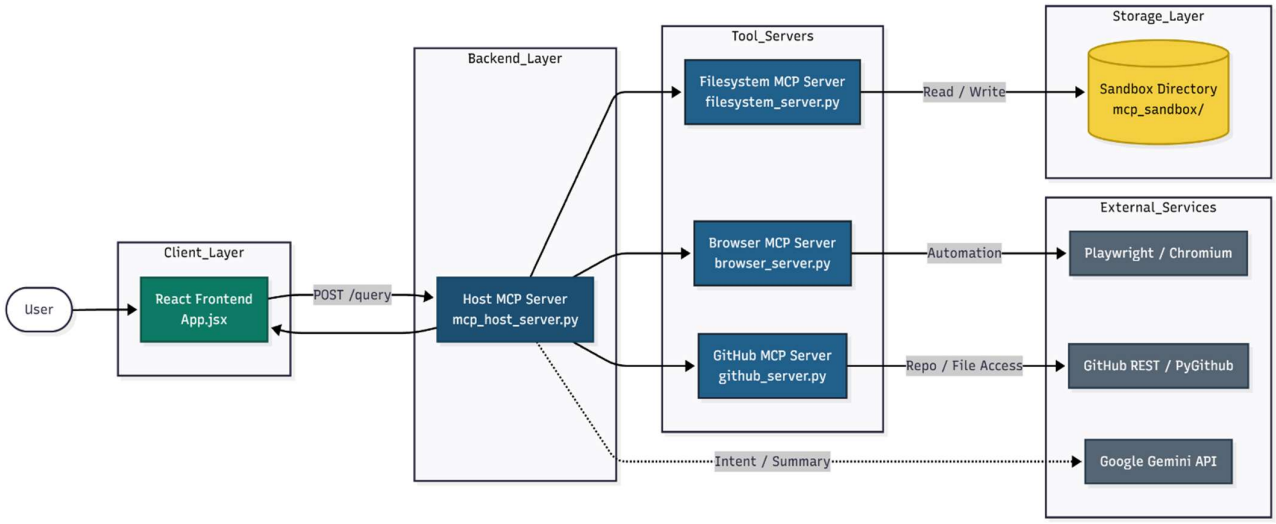


Figure 3.2: System Architecture of the Unified MCP Framework

The central point of the Unified MCP Framework is the Host MCP Server which acts as the central or the main part , conductor that determines the information flow in and out of all the system components. The frontend makes queries to the Host MCP Server which are expressed as natural language and at this point the server starts to understand the intent of the user via internal heuristic or optional support of the Gemini model. It uses this knowledge to decide which dedicated MCP server would be most appropriate to respond to the request.

Once the correct component is identified , the Host MCP Server sends the execution request, waits the tool response, processes the output into a structured format and sends the end result , which is the final result consolidation, back to the front end. This coordination mechanism will be used to make sure that the Filesystem, Browser and GitHub MCP Servers are coordinated to not act as distinct components but as a single smart agent that provides continuity between different environments.

The Filesystem MCP Server is specifically used in managing file related functions in a highly regulated and isolated environment. It will only execute within the sandbox directory that was created when the system was started and therefore, any form of unintended access cannot take place outside this secure domain. Path validation is done by the server which resolves and sanitizes path names and makes sure that no file path traversal or no unauthorized directory jumps can take place. After authentication, the server becomes read / write , allows file appending, directory creation , directory listing and search of files with patterns based on glob patterns.

Any write operation is atomically ımplemented so as to prevent partıal wrıtes or ınconsıstency wıth the exıstıng information and hard size constraints and safety restrictıons ensure data ıntegrıty wıthın the sandbox. These measures have made the Fılesystem MCP Server a secure fıle manıpulation server that meets the basic need of security.

Another essential component is the Browser MCP Server which is aımıng at connectıng to lıve web envıronments usıng Playwrıght automatıon engıne. The server is charged with the responsibilıty of carryıng out browser based actıvities, which include searching and viewing webpages , scraping visible text and scraping critical metadata. It loads pages and inspects them programmatıcally usıng a headless Chromıum browser wıth extreme restrictions to guarantee safe and deterministic behavıour . Browser MCP Server does not run scrıpts, does not do ınteractıons that may make changes to external content or vısits dangerous domains. Rather, it is used in controlled and read-only mode in which information retrieval is the main activity. These precautions render it appropriate ın supportıng research focused jobs, valıdatıon of user commands and textual ınformatıon that the Host MCP Server may require in further interpretation.

GitHub MCP Server can be used to make secure communication with remote repositorıes eıther through the PyGıthub lıbrary or wıth a REST-based fallback ımplementatıon. This component supports a number of repository-level operations , including retrieval of dırectory structure , ınspectıon of metadata , fetchıng file content and validation of repository settings. A GitHub Personal Access Token with the least amount of required scopes is used to perform authentication and ensure that only a few access attempts are allowed. The server fully complıes wıth guıdelınes of the GıtHub API , handles rate lımıts and does not modıfy the information of the repositories and works in fully read-only mode to ensure that the information is inspected. Wıth thıs feature, the framework allows developers to navıgate reposıtories, inspect code files and extract structural information without having to browse GıtHub ınterfaces.

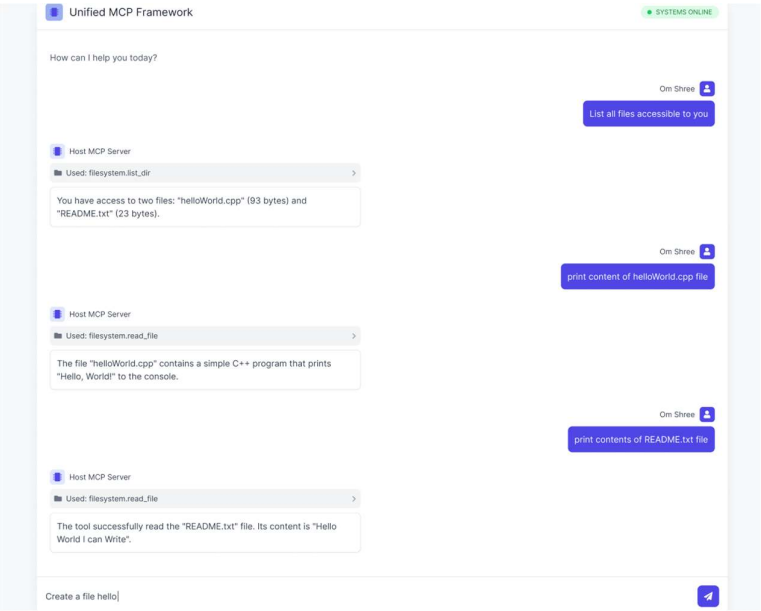


Figure 3.3: MCP Frontend UI

The frontend part of the system ıs the user-facıng interface and it is implemented with React and Vite ın terms of the best buıld tıme. It offers a chat-lıke ınteractıve envıronment, ın whıch developers can feed querıes and vısualıze the detaıled responses generated by the backend. Besıdes showıng the final summarized output , the frontend also shows execution traces , raw tool responses and intermediate states to enable users to view how exactly the framework handles each request. This visibility enables developers to know how MCP based agents behave and can aid in debugging, assessment and instruction. The frontend keeps the state in the course of the browsing session , which avoids any persistent storage of data. Its simplicity and emphasis on clarity render it appropriate to illustrate how tool-based AI processes can be made to work in real time.

3.3 DATA PREPARATION

The data preparation of this project aims at establishing the working functionality that Unified MCP Framework needs in order to operate effectively. Given that this system is not built on a conventional machine-learning workflow and also relies not on large datasets , the preparation part focuses on setting up environments, setting up secure directories , installing tools needed to automate browsers, setting up access to GitHub and making sure that the frontend can adequately respond to the tools. This stage is intended to provide a solid and predictive base on which every part of the framework can communicate with another without contradictions and unforeseen actions.

The preparation also makes sure that all the settings of the Host MCP Server , Filesystem MCP Server, Browser MCP Server and GitHub MCP Server start with the right settings. All the components of the system have their own form of operational data and the process of making ready these parts is a significant step towards the functionality of the framework as an integrated whole.

A. Configuration Data

The initial phase of preparation entails preparation of configuration data to initialize the backends. This contains environment files, server settings and tool registration information. The backend needs a .env file, the contents of which include the Gemini API key and in case required, a GitHub Personal Access Token. These values enable the system to authenticate safely with third party services. FastAPI backend is also set up with CORS settings in such a way that the React frontend can be able to communicate with it without any limitations.

Besides this, every MCP tool server should enroll its available tools with the Host MCP Server. Information concerning the names of tools, format of parameters and outputs are received by the host through this registration process. By preparing this metadata, it is possible to make sure that the host is able to correctly interpret and send user requests to the relevant tool server.

B. Sandboxed Filesystem Data

The Filesystem MCP Server is served on a controlled directory called mcp_sandbox. This directory is generated in the setup process and is meant to separate all the file operations to the rest of the computer. This sandbox is prepared by setting up the base folder structure and ensuring that the system only limits file accessibility to this directory and not any other directory.

At this stage, safe-path rules are used to enforce that unauthorized access is not done using the absolute paths or that they do not access the directory outside the sandbox. Reading and writing file data limits are also defined to prevent the performance problem when running tools. Some sample files can be found within the sandbox in order to test reading , writing , appending and listing operations. This test ensures that the Filesystem MCP Server is well mannered and predictable during the project

C. Browser Data Preparation

Browser MCP Server is based on Playwright and a Chromium browser instance. The installation of all the necessary browser binaries and checking that automated browsing is functioning properly are a part of preparing this environment. Simple algorithms to search operations , and to extract visible text on web pages are put to test in this stage.

Trial runs will be done on standard web pages and search queries in order to ensure that the server gives stable output when parsing headings and page content. The value of timeouts and error management is also verified and this is to ensure that automation of a browser does not get trapped on a slow or unresponsive site. This preparation is necessary in order to be sure that the Browser MCP Server is prepared to make reliable and secure web interactions upon being called by the system.

D. GitHub Tool Preparation

GitHub MCP Server must be authenticated and granted access to a repository before it can be able to do file retrieval or metadata extraction. In case a GitHub Personal Access Token is made , the system sets up the PyGithub client to verify requests in an encrypted way. It will allow avoiding rate limits and will grant access to personal repositories when needed.

A set of repositories is chosen to be tested during preparation. The server is tested on its capability to obtain file structures, decode file contents and retrieve metadata on repositories. The fallback policy of unauthenticated public repositories is also tested. This makes the system dynamic and the GitHub MCP Server is functional in both the authenticated and unauthenticated states.

E. Frontend Data Handling

The React frontend is ready to take user input , save temporary responses of the tools and display the execution traces in a clear manner. The frontend is not a permanent data-store ; although it retains the state throughout the duration of the session. These are user requests , tool output , summaries and logs by the Host MCP Server.

Tests on preparation consist of making the frontend receive multiple consecutive requests and making sure that when the interface is used it does not overwrite previous traces of the tools and the idea that the message flow is easy to follow. All these tests ensure that the frontend is prepared to communicate with all backend servers and to display the results in a structured and comprehensible manner.

3.4 IMPLEMENTATION

The Unified MCP Framework implementation unites a number of coherent elements that interact to give the user, AI model and tool servers the ability to work together with the environment. The four most important devices in the system include the Host MCP Server, the Filesystem MCP Server, the Browser MCP Server and the GitHub MCP Server. All these servers have a purpose well defined but they work closely with each other. Upon the user making a request with the help of the frontend, the Host MCP Server will understand and process the natural-language input and decide what server or tool to forward the request to. This enables the system to act as a smart assistant who is capable of comprehending tasks, executing the necessary tasks as well as the provision of pertinent results.

One of the most important elements of the implementation is the attention to the safety, reliability and controlled implementation of the tool. The Filesystem MCP Server is fully contained in a specific sandbox folder so that file operations are not integrated with the host computer. It authenticates every route, does atomic write operations, deals with directories and reads files with an absolute prohibition of unauthorized access to any object outside of the sandbox.

The Browser MCP Server is developed based on Playwright and headless Chromium engine, which enables the system to process search queries, access webpages and retrieve the content that is visible on them in the structured form. Any interaction with the browser is limited to the read-only mode to ensure the absence of unsafe actions that can be initiated accidentally. In the same way, the GitHub MCP Server communicates with GitHub via authenticated API requests and identifies operations like exploration of repository organization, file retrieval and metadata gathering. It does not do any write or modify operations and also interact with GitHub is safe and predictable.

Host MCP server , which is implemented with FastAPI , takes care of organizing the whole working process. It loads tool servers at setup, operates asynchronous execution pipelines , and formats the ultimate responses back to the user. It also contacts the Gemini API when needed to create summaries or do intent clarification. The Host MCP Server combines the results of the Filesystem , Browser and GitHub servers into a single formatted response and lets users interact with the system as if they are browsing and reading irrespective of the nature of the operation they choose to perform. This layer enables the system to act as a single system as opposed to a set of individual tools.

The frontend interface is the final component of the system . It allows to provide the user with a transparent and user-friendly platform where they can enter queries and receive results, as well as examine tool-level execution information. It is developed in React and Vite so that it could load quickly , provide a good experience in handling state or even responding to it. Both the summarized and the raw responses of the tools are displayed in the interface and it assists users to know how the system arrived at the answers as it did. This is because of its nature ; the frontend is lightweight and based on sessions, which makes the framework ensure that a user is in a position to interact with the system in a natural way with no delays or interruptions.

On the whole, the Unified MCP Framework implementation reveals that various specialized servers, central orchestrator and a modern frontend can collaborate to provide a single and smart developer-assistance system. The safety controls, modular structure and structured communication combination guarantee that the framework is resilient, transparent and appropriate toward real-world workflows.

3.4.1 Host MCP Server

Host MCP Server is the focal point of coordination of the whole structure. It is executed with FastAPI and it is in charge of loading all the available MCP tool servers at start time, such as the Filesystem, Browser and GitHub servers. Upon initiation, the Host Server keeps a database of capabilities of the tools and presents only one endpoint that accepts user queries via the frontend. The server receives a request and based on the user input it would interpret the input of the user and determine whether the request is to execute a tool or a direct model response and forward the request to the relevant backend component. This is because this method allows the system to behave the same way even in the presence of different tasks like file manipulation, web page extraction, repository inspection or content summarization.

In addition to routing work , the Host MCP Server handles all the communication between the user interface, the tool servers and optional processing (LLM-based). Once a tool has completed its execution, the Host Server retrieves the raw output, formats it in a predictable form as a JSON and when necessary pushes the result over to the Gemini API to be summarized or refined. This is done to guarantee that the user does not only get the raw data out of the tool but also a clear and comprehensible interpretation of the result. By providing such a multi-layered workflow, the Host MCP Server remains reliable, transparent and maintains a constant interaction with all components of the Unified MCP Framework to ensure that the whole system functions as a single cohesive unit.

A. Server Initialization (Excerpt)

```
# Create the main FastAPI application for the Host MCP Server.
# The 'lifespan' function will run automatically when the server starts.
app = FastAPI(title="MCP Host Agent", version="1.0", lifespan=lifespan)

@asynccontextmanager
async def lifespan(app: FastAPI):
    # Load the active Gemini model from environment variables (.env file)
    global ACTIVE_MODEL_NAME
    ACTIVE_MODEL_NAME = configure_genai_from_env()

    # Load and register all connected MCP tools (Filesystem, Browser,
GitHub)
    load_tools()

    # Hand control back to FastAPI after initialization is complete
    yield
```

Figure 3.4: Host MCP Server Initialization

The Host MCP Server is set up by this code. When the FastAPI application starts , it will load the currently active Gemini model from the environment using the lifespan function. All MCP tools will be registered (Filesystem , Browser , GitHub) at this time , which completes the configuration of the Host MCP Server and makes it possible to route incoming user queries from the Host MCP Server to all other connected Host MCP Servers.

B. Query Handling and Tool Execution

3.4.2 Filesystem MCP Server

```
@# API endpoint that receives the user's query from the frontend.
# It returns a HostResponse containing the final answer and tool execution
trace.
@app.post("/query", response_model=HostResponse)
async def process_user_query(query: HostQuery):

    # Collect all tools available across connected MCP servers
    # (Filesystem, Browser, GitHub).
    available_tools = []
    for name, srv in CONNECTED_SERVERS.items():
        available_tools.extend(srv.list_tools())

    # Ask the LLM (Gemini) to choose the best tool for the query.
    decision = await llm_select_tool(query.user_query, available_tools)

    # If the LLM cannot decide, fall back to a simple rule-based selector.
    if not decision:
        decision = heuristic_select_tool(query.user_query, available_tools)

    # If a valid tool decision was made:
    if decision:
        # Identify which MCP server the selected tool belongs to.
        server = CONNECTED_SERVERS[decision["server_name"]]

        # Run the selected tool with its arguments.
        result = await server.run_tool(decision["tool_name"],
decision.get("args", {}))

        # Generate a summarized final answer for the user.
        final_text = await generate_final_answer(query.user_query, result)

        # Return the final response along with tool execution details.
        return HostResponse(
            final_answer=final_text,
            tool_calls_executed=[{
                "server": decision["server_name"],
                "tool": decision["tool_name"],
                "args": decision["args"],
                "result": result
            }]
        )
```

Figure 3.5: Tool Routing Function

The MCP Server combination of the Filesystem , Browser , GitHub will evaluate the Gemini model (or a heuristic fallback) to choose which MCP tool will perform the operations necessary to return a response to the user query. Upon completion of the execution of the defined tool , an answer will be generated and a list of all operations performed by the tool will be provided to the user alongside the final answer.

A. Safe Path Handling

```
def _get_safe_path(relative_path: str) -> Path:
    # Normalize the input path: remove extra spaces and unify slashes
    rel = relative_path.strip().replace("\\", "/")

    # Reject absolute paths and parent directory escapes (e.g., "../")
    if os.path.isabs(rel) or ".." in rel.split("/"):
        raise ValueError("Invalid path")

    # Convert the sanitized relative path into an absolute path inside
the sandbox
    candidate = (SANDBOX_DIR / rel).resolve()

    # Ensure the resolved path is still within the sandbox directory
    if not str(candidate).startswith(str(SANDBOX_DIR)):
        raise ValueError("Outside sandbox")

    # Safe path approved
    return candidate
```

Figure 3.6: Secure Path Validation

To limit all operations related to file systems to the mcp_sandbox directory , the function will take steps to prevent absolute path references , path traversal attempts through the “..” sequence , and ensure the path that is resolved back into the mcp_sandbox remains inside of the sandbox. As a result , there are no chances for unauthorized files or directories to be accessed by the Filesystem MCP Server.

B. Example: Read File Tool

```
elif tool_name == "filesystem.read_file":
    # Validate and convert the provided file path into a safe sandboxed
path
    p = _get_safe_path(args["path"])

    # Open the file in read mode and read only up to MAX_READ_CHARS
    with p.open("r", encoding="utf-8") as f:
        txt = f.read(MAX_READ_CHARS)

    # Return the file's original path and its extracted content
    return {"path": args["path"], "content": txt}
```

Figure 3.7: Read File Tool Logic

The Filesystem MCP Server includes the tool filesystem.read_file , which first verifies that the path requested has been validated by _get_safe_path() against the secure sandbox , then Open()s the file to read only a limited number of characters from it , and returns the resulting text back to the Host MCP Server.

C. Example: Write File Tool

```
elif tool_name == "filesystem.read_file":
    # Validate and convert the provided file path into a safe sandboxed
path
    p = _get_safe_path(args["path"])

    # Open the file in read mode and read only up to MAX_READ_CHARS
    with p.open("r", encoding="utf-8") as f:
        txt = f.read(MAX_READ_CHARS)

    # Return the file's original path and its extracted content
    return {"path": args["path"], "content": txt}
```

Figure 3.8: Write File Tool Logic

The Filesystem MCP Server also implements tool filesystem.write_file. After validating the path against the sandbox , it writes out the file content to a temporary file first to make sure that all writes are atomic , and then moves that file into place after it's complete. This ensures that there will never be any partial writes in the sandbox , and that all file operations are secure and reliable within this environment.

3.4.3 Browser MCP Server

The Browser MCP Server is a Playwright application that is used to form the interface between the system and the web through the ability to perform automated browsing. It is set to carry out actions like search queries, clicking on webpages and acquiring the visible content of the text in the form of a structured data, but with a

very tight safety margin. Once the Host MCP Server decides that a user request needs online information, it calls the Browser Server which in its turn starts a headless Chromium instance and uses it to run the request.

Only readable and harmless content is extracted by the server without involving any scripts, dynamic interactions or actions that may alter the external pages. The Browser MCP Server allows the framework to use real time web information in the workflow by providing clean and structured text output, which allows the workflow to be fully traceable and predictable and safe.

The server additionally normalizes all browser responses in order to have uniformity in relation to varied queries and web sites. Its secured setting denies entry to harmful URLs , pop-ups , or interactive features that may disorient the automated processes. This renders the Browser MCP Server as a sound element towards the collection of real time data without compromising on system integrity.

Google Search

```
async def _search_google(self, query: str, count: int = 8):
    # Encode the search query so it is safe to use in a URL
    safe_query = quote_plus(query)

    # Open a new headless browser page using the Browser MCP Server
    page = await _browser_manager.new_page()

    # Navigate to Google Search with a timeout limit
    await page.goto(f"https://www.google.com/search?q={safe_query}",
timeout=20000)

    # Select all <h3> elements, which usually contain search result
titles
    elements = await page.query_selector_all("h3")

    results = []
    # Extract titles and their corresponding links for the top 'count'
results
    for el in elements[:count]:
        title = (await el.inner_text()).strip()
        href = await el.evaluate("node => node.closest('a')?.href")
        if title and href:
            results.append({"title": title, "link": href})

    # Return a structured response for the Host MCP Server
    return {"engine": "google", "results": results}
```

Figure 3.9: Google Search Tool

This code is the implementation for the Google search tool in the Browser MCP Server. It takes the search query and formats it properly for use with Google , opens a headless Chromium browser and searches Google for the formatted query. Then it reads back the titles and links from the first N results and sends back the clean structured results in a manner that they can be used for further processing or summarizing by the Host MCP Server.

3.4.4 GitHub MCP Server

The GitHub MCP Server has the responsibility of controlling and authenticated access to remote repositories and thus the framework can make actions like reading files , browsing directory strucures and retrieving metadata. This server runs on the PyGithub library in cases where an authenticated Personal Access token is present by which the system may make secure API requests without exceeding GitHub rate limits or permission boundaries.

Where the token is not availed , the server will be able to fall back to basic REST API requests to get public repositories, so vital functionality is still available. The GitHub MCP Server captures this logic and offers a standardized and predictable interface with which to interact with the resources of GitHub resources regardless of their visibility or authentication status. One of the important functions that have been deployed in this server is the possibility to access the contents of a given file in a storage facility in a secure manner.

The following snippet illustrates the way that the system will discover the target repository, resolve the file path and decode the base64-encoded information that is being returned by the GitHub API. This enables the Host MCP Server to visualize code files, documentation and configuration resources as part of workflow of the user. The limited access token scopes and the strict validation of the access token are also useful in ensuring that all repository interactions are safe and read-only with no possibility of accidental changes or unauthorized operations and ensuring that the access is seamlessly integrated with the remote version-controlled resources.

Read File from Repository

```
if tool_name == "github.read_file":
    # Connect to the target GitHub repository using the provided full
repo name
    repo = client.get_repo(args["repo_full_name"])

    # Retrieve the file object from the repository at the given path
    content_file = repo.get_contents(args["path"])

    # GitHub returns file contents in base64, so decode it into plain
text
    raw = base64.b64decode(content_file.content).decode("utf-8")

    # Return the file path, the file's SHA reference, and its decoded
content
    return {"path": args["path"], "ref": content_file.sha, "content":
raw}
```

Figure 3.10: GitHub Read Tool

The provided code demonstrates how to consume an API which allows interaction with a GitHub repository by allowing a client to issue requests to read a specific file in a GitHub repository. The method used to implement this functionality is by sending an HTTP GET request to the GitHub API with the filename path specified, decoding the contents of the retrieved file (which is returned as a base64 encoded string), and responding with the decoded text of the file plus its SHA reference. Through this method, a client can easily access files in a GitHub repo through an API.

3.4.5 Frontend Implementation (React + Vite)

The frontend provides the chat interface, sends queries to the backend and displays tool execution traces.

Sending Query

```
// Send the user's query from the frontend to the Host MCP Server
const response = await fetch("http://127.0.0.1:8000/query", {
  method: "POST",
  headers: { "Content-Type": "application/json" },

  // Include the user's message and session ID in the request body
  body: JSON.stringify({ user_query: input, session_id: "default" })
});

// Convert the server's response into JSON
const data = await response.json();

// Add the final answer from the Host MCP Server to the chat messages
setMessages(prev => [...prev, { role: "assistant", text:
data.final_answer }]);

// Store the tool execution trace so the frontend can display how the
request was processed
setTrace(data.tool_calls_executed);
```

Figure 3.11: Frontend API Call

The second section of the code provides a demonstration of how a user can submit queries via HTTP POST requests to the Host MCP server using the React front-end client. The Host MCP server will process these requests, return the final answer, and provide logs of tool usage. With this method of communicating between the front-end UI and the back-end MCP server, it establishes a complete chat-based workflow.

3.5 KEY CHALLENGES

A number of technical and architectural issues were experienced in the development of the Unified MCP Framework. These predicaments occurred because of integration of various asynchronous tools , heavy security concerns and interoperability of backend components. Following are the key issues that were faced in the project and how they have been handled.

1. Safe Tool Execution.

Among the key design considerations was to make sure every tool (Filesystem, Browser and GitHub) was working within very rigid security concerns. The Filesystem MCP Server had to stop the accidental or deliberate access to the system directories not within the sandbox.

Resolution:

An effective path-sanitization system was adopted through resolve() test, elimination of traversal sequences of the form of a dot and maintenance of directory confinement within mcp_sandbox/. This made all the file operations completely isolated and secure.

2. Managing Multiple Server Asynchronous Operations.

The Browser MCP Server and the Host MCP Server have strong dependency on the execution of asynchronous functionality through the supports of asyncio and Playwright. Co-ordination between concurrent browser contexts, page rendering and tool invocation was important.

Resolution:

A Browser Manager centralized class was also used to make browser instances and manage them. The locking mechanism was shared to make sure that the browser was developed once and re-used effectively to prevent synchronization errors and redundant overheads.

3. Integrating Multiple External APIs.

The project uses Google Gemini in natural language understanding and summarization.

For browser automation it uses built-in playwright and PyGithub library and GitHub REST API.

Both APIs have their rate limits, authentication and structure of responses and it is not a trivial task to coordinate them.

Resolution:

Each MCP server was separated into independent modules with the help of a modular architecture. The Host MCP Server was used to discover tools, route and manage errors ; therefore, the separation of responsibilities was kept.

4. Presenting Tool Trace Visibility.

Another project requirement was to demonstrate to the users how the various tools were called and raw JSON results. This traceability feature was to be designed with proper planning in order to ensure that the frontend is not overloaded.

Resolution:

Each tool call sent by the system contains:

- server name
- tool name
- arguments
- tool output

This enabled the frontend to display clean structured traces of the tools without storing or exposing unneeded system-level data.

5. Handling Errors Gracefully

Frustrating browser navigation, inability to use GitHub or manipulate files might lead to disruptions or partial responses.

Resolution:

Error-handling mechanisms were made consistent throughout the servers. Each tool provides a structured dictionary consisting of HTTP-like codes (200, 400, 404, 500) which the Host MCP Server can understand as long as there is a proper error message to give meaningful advice to the user.

6. Striking a balance between Flexibility and Security.

MCP promotes very strong tool usage yet Unlimited access (played by the Filesystem and Browser tools) might become a security threat.

Resolution:

Tough measures were imposed which included:

- outlawing insecure URLs (localhost, internal IPs)
- Allowing only access to sandboxed directories.
- Restricting maximum reads and writes as well.
- Checking of GitHub repository inputs before processing.

These were measures that the system was not compromised in terms of functionality.

7. Ensuring Proper Frontend Backend Communication.

The React frontend interacts with two systems that are independent of each other : the Host MCP Server and each tool. A stable API response and avoidance of CORS problems entailed changes to the backend.

Resolution:

The FastAPI server was configured with proper CORS settings (controlled origin localhost:5173) to allow the front-end and the server to communicate smoothly.

CHAPTER 4: TESTING

4.1 TESTING STRATEGY

The appropriate, reliable and stable Unified MCP Framework was achieved by a structured and systematic method of testing. As the system is made up of several independent but interacting components (Frontend UI , FastAPI Host Server , Filesystem MCP Server , Browser MCP Server and GitHub MCP Server), testing was conducted in several layers. The testing plan consisted of unit testing, integration testing, functional testing and manual scenario testing.

1. Individual MCP Server Unit Testing.

All the MCP servers (Filesystem, Browser, GitHub) were tested separately, then fully integrated.

- Tests of filesystem -: check that path sanitization, file reads/writes within the sandbox, creation and deletion of directories, extracting metadata.
- Browser -: Testing Playwright start , Google search, page text retrieval, metadata retrieval, link retrieval and screenshot acquisition.
- Github testing -: ensuring that a GithHub repository is listed, file read, branch manipulation, issue creation and commit access work with both PyGithub and fallback to REST.

Unit testing was used to make sure that all the tools acted appropriately and provided structured results in the desired format (dictionary with code, result or error).

2. Integration Testing and Host MCP Server.

Once the separate elements were verified, the MCP Host Server was put to test, in order to verify:

- correct loading of tools
- proper execution request routing of tools.
- effective performance of the asynchronous operations.
- accurate JSON responses .
- proper error propagation .
- Multi-step scenarios involved in the integration testing were:
- user query - selection of the tool followed by execution and summarization of the tool by the LLM.
- user query without using a tool then straight LLM answer.
- The tool calls in one session several times.

3. Frontend End-to-End Testing

React frontend was applied to test the entire workflow as a user. This validated -:

- real-time submission of requests.
- online/offline connectivity (backend)
- tool trace visualization
- response rendering
- adequate error messages when there is failure.

Frontend assisted in verifying real-life latency, usability and stability with real-world input patterns.

4. API Testing

Endpoints of FastAPI (/query, /tools, /health) were tested with such tools as:

- Postman
- Thunder Client (VS Code)
- GET/POST testing using browser.

This made sure that the API would respond with the right HTTP codes, anticipated JSON structures and gracefully deal with invalid inputs.

5. Security and Boundary Testing.

As a result of availability of tools that can access files and automate browsers, testing associated with security was needed.

Key checks included:

- blocking unsafe URLs (localhost, local IP addresses)
- size write restriction.
- checking GitHub repository names formats.
- providing confinement of sandbox in filesystem tool.
- authenticating the error messages of invalid and blocked actions.

These tests were to ensure that the system was safe and predictable when it received improper or malicious input.

6. Real-World Scenario Testing (manual):

This is the method that is employed to test the system in a real-life situation.

The system was also tested using real user queries to validate practicality. Examples included:

- " Search for the latest cloud security updates "
- " List my GitHub repositories "
- " Show me the metadata of example.com "
- " Create a directory and write a file inside it " -: Real world usability and correctness were tested manually.

4.2 TEST CASES AND OUTCOMES

The testing process was focused on validating the core functionality of all MCP tools (Filesystem, Browser, GitHub) and ensuring proper coordination through the Host MCP Server. The following tables summarize the essential test scenarios executed during the project.

Table 4.1: Backend MCP Server Test Cases

Test ID	Component	Description	Expected Outcome	Result
FS-01	Filesystem	Create directory	Directory created successfully	Pass
FS-02	Filesystem	Read & write file	File written and content read correctly	Pass
FS-03	Filesystem	Block unsafe path	Error for path traversal	Pass
BR-01	Browser	Google search query	List of title–URL results	Pass
BR-02	Browser	Extract page text	Visible text extracted	Pass
BR-03	Browser	Block unsafe URL	Error for internal URL	Pass

GH-01	GitHub	Read file from repo	File content returned accurately	Pass
GH-02	GitHub	Invalid repo test	Proper error returned	Pass

This table 4.1 summarizes the primary backend tests covering all three MCP servers. The Filesystem server was verified for safe path handling, file creation and data retrieval. The Browser server was validated for search execution and content extraction using Playwright. GitHub server tests confirmed correct interaction with repositories and correct handling of invalid inputs. All expected behaviours matched actual outputs, confirming stable and secure tool functionality.

Table 4.2: Host MCP Server Integration Test Cases

Test ID	**User Query Type**	**Expected Tool**	**Actual Behaviour**	**Result**
HS-01	File creation request	filesystem.write_file	Tool selected and executed correctly	Pass
HS-02	Web research query	browser.search	Search results returned	Pass
HS-03	GitHub inspection query	github.read_file	File content retrieved	Pass
HS-04	General knowledge query	No tool (LLM only)	Direct answer generated	Pass

Integration tests validated the Host MCP Server's ability to interpret natural-language queries, route them to the correct tool and return meaningful results. The tool-selection logic (LLM-assisted and heuristic fallback) behaved as designed. Both tool-based and non-tool queries produced accurate responses, confirming reliable orchestration.

Table 4.3: Frontend Application Test Cases

Test ID	Description	Expected Outcome	Result
FE-01	Send chat query	Backend response displayed	Pass
FE-02	View tool execution trace	JSON trace shown correctly	Pass
FE-03	Backend offline	"Backend Offline" indicator visible	Pass
FE-04	Handle long output	UI scrolls properly without overflow	Pass

Frontend tests focused on validating user interaction, backend communication and UI stability. The application correctly displayed backend responses and tool execution traces. In failure scenarios, such as backend unavailability, appropriate status indicators were shown. The interface remained responsive even for long outputs, demonstrating robustness and usability.

CHAPTER 5: RESULTS AND EVALUATION

5.1 RESULTS

The Unified MCP Framework developed in this project successfully demonstrates a functional multi-tool AI agent system capable of executing real developer-oriented tasks. The results confirm that the system meets all defined objectives , including natural-language understanding , tool orchestration , secure execution and structured response generation.

1. **Successful Tool Invocation Through Natural Language**

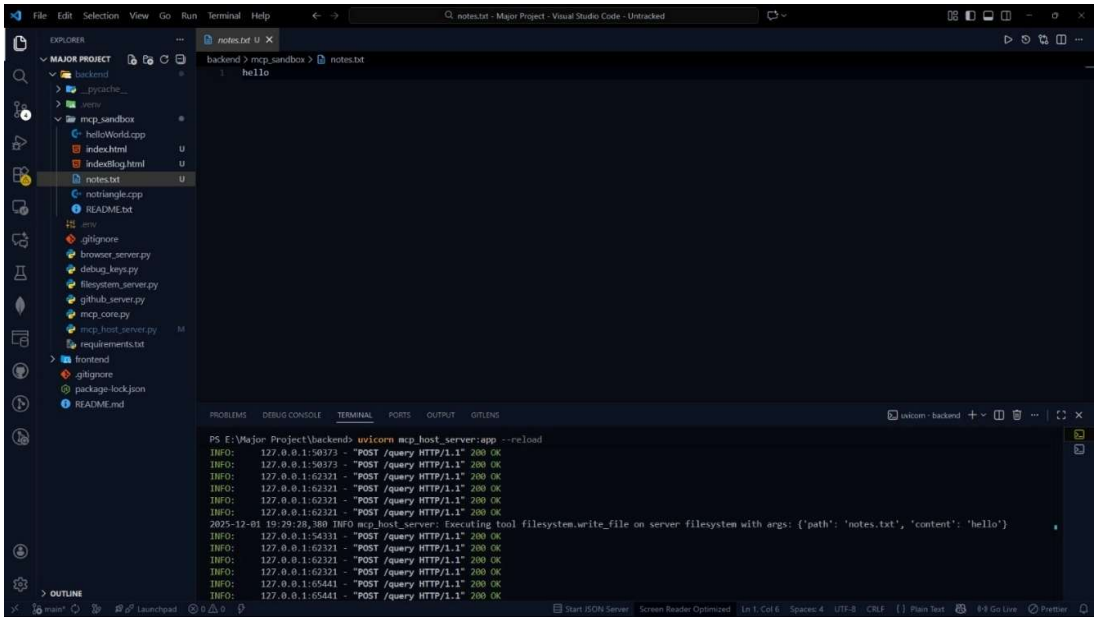


Figure 5.1: VS Code Sandbox File Creation

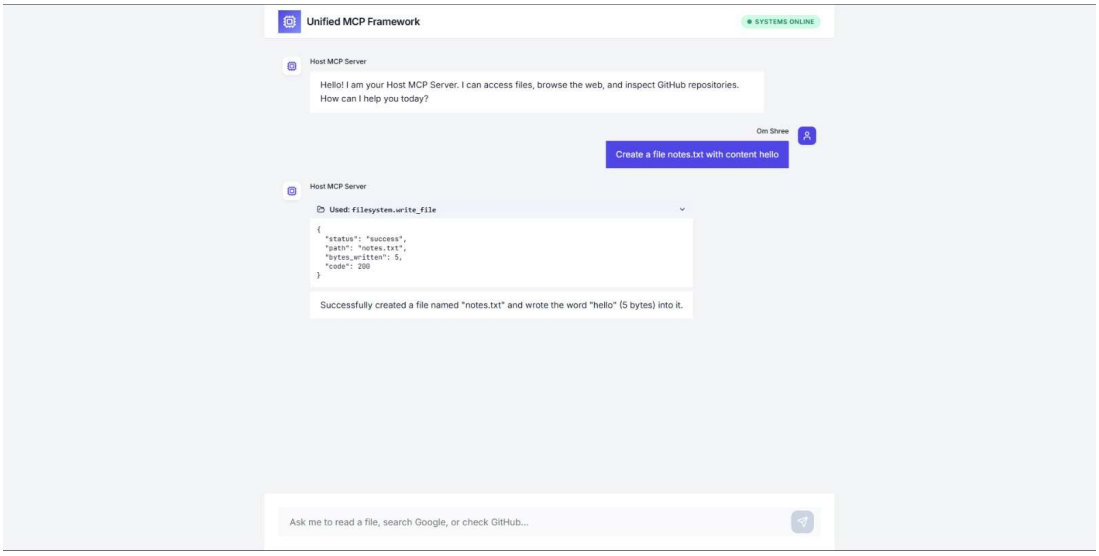


Figure 5.2: MCP Frontend Tool Execution Output

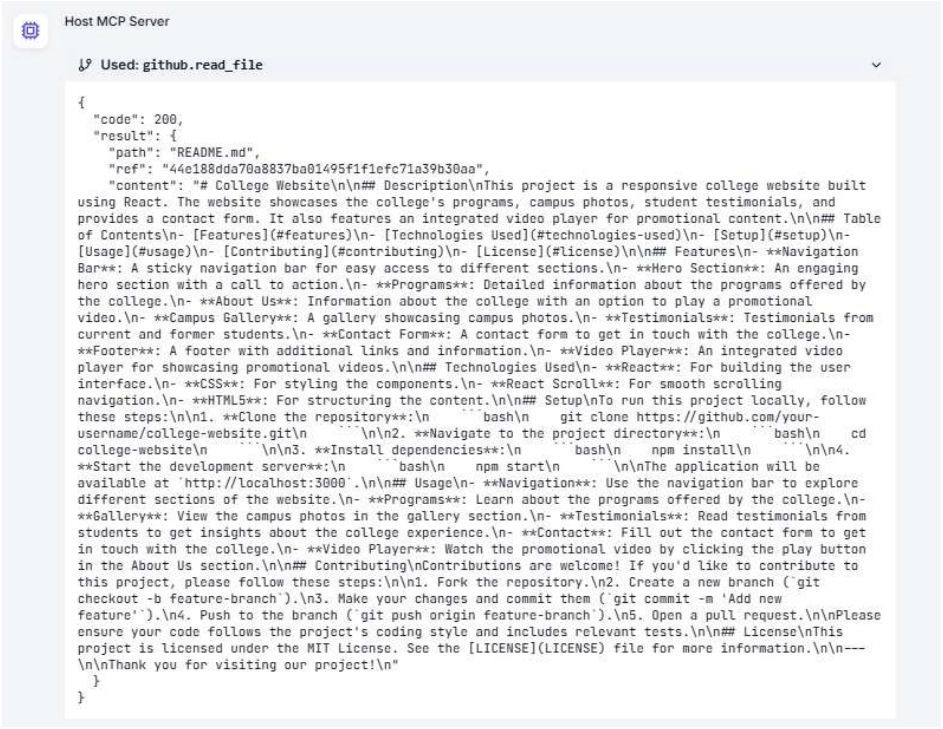


Figure 5.3: Tool call

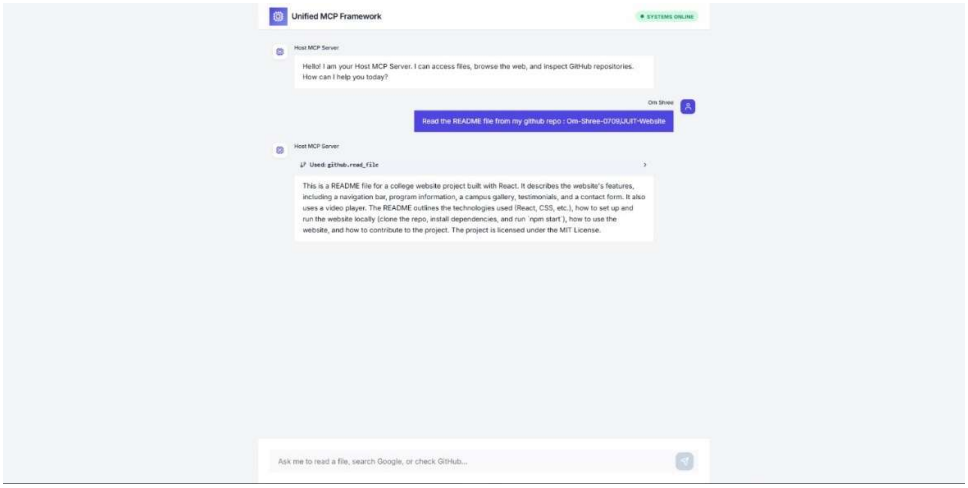


Figure 5.4: Result

The system accurately interpreted user queries and selected the appropriate tool. Examples of working scenarios:

- Query -: “Create a file notes.txt with content hello” ,
  Tool invoked -: filesystem.write_file
  Result -: File created successfully in sandbox
- Query -: “Read the README file of my GitHub repo”
  Tool invoked -: github.read_file
  Result -: Returned exact file contents

These results validate the correctness of the Host MCP Server's tool-selection and routing logic.

2. Secure and Isolated Filesystem Operations

All file operations executed strictly inside the sandboxed directory. Attempts to perform invalid or unsafe operations (e.g., accessing "C:\", "/etc/passwd", using "../") resulted in controlled errors. This confirms successful implementation of the path-sanitization logic.

3. Reliable Browser Automation

Playwright-based automation produced consistent results:

- Google/Bing search queries returned relevant results
- Webpage text extraction worked on multiple domains
- Unsafe URLs (localhost, internal IPs) were correctly blocked
- Invalid links produced proper error messages

This demonstrates robust asynchronous browser functionality.

4. Accurate GitHub Repository Interaction

Using PyGithub and REST fallback , the system successfully:

- Fetch the file contents
- List all the repositories
- Handle the invalid repository names safely
- Decode the base64-encoded GitHub file content correctly

These outputs confirm correct integration with GitHub APIs.

5. Performance Observations

- Tool calls executed quickly under normal conditions.
- Browser operations were comparatively slower (expected due to live page rendering) .
- Memory footprint remained low during typical use .
- No crashes or unhandled exceptions were observed during prolonged testing .

Overall, the system met all planned objectives. All major components performed reliably and consistently under various test conditions, demonstrating a well-designed and secure multi-tool AI agent framework.

5.2 COMPARISON WITH EXISTING SOLUTIONS

This section compares the proposed system with existing approaches in multi-tool AI agent frameworks.

1. Comparison with Traditional Chatbots

Traditional chatbots provide only text-based responses and cannot execute dynamic operations such as filesystem manipulation or real-time web browsing.

Table 5.1: Comparison between Chatbots

Feature	**Traditional Chatbots**	**Proposed System**
Natural language understanding	Yes	Yes
File operations	No	Yes
Web browsing	No	Yes
GitHub integration	No	Yes
Tool execution trace	No	Yes
Sandbox security	Not applicable	Yes

The proposed system provides much higher capability and real utility for software development tasks. Another important distinction is the system's ability to maintain continuity across different types of tasks, something traditional chatbots cannot provide. While chatbots can answer questions or generate explanations, they typically fail when users require direct interaction with files, live webpages or version-controlled repositories. The Unified MCP Framework bridges this gap by giving the agent operational capabilities rather than limiting it to conversational responses. This transforms the system into an actual development assistant rather than a passive generator of text, making it significantly more useful for real engineering workflows.

2. Comparison with Cloud-Based LLM Tool Platforms (e.g., LongChain Agents)

Cloud-based agent frameworks offer tool execution but typically rely on cloud-hosted runtimes, lack strict local sandboxing and require continuous internet connectivity.

Table 5.2 : Aspects of Frameworks

Aspect	**Existing Agent Frameworks**	**Proposed System**
Offline capability	Limited	Backend can run locally
Sandbox enforcement	Weak	Strict path and tool isolation
Browser automation	Not always available	Fully supported via Playwright
Frontend transparency	Minimal	Full tool-trace visibility

The suggested system offers excellent capability and actual usefulness in the software development work. The other notable difference is the ability of the system to have continuity across the various kinds of work, which traditional chatbots cannot deliver. Although chatbots are able to respond to questions or create explanations, they usually do not work when users need to work with files, live webpages or controlled repositories. The Unified MCP Framework is a step to fill this gap by providing the agent with the operational capabilities instead of reducing it to conversations.

This is what turns the system into a literal development assistant and not a text-generating passive engine, and thus much more applicable to actual engineering processes. The suggested system offers higher security assurance and increased control on the execution environment. The proposed system has a more predictable and customizable execution model in addition to better security. Remote servers in cloud-based solutions can potentially provide latency, unreliable performance or occasionally a blockage of API access.

3. Comparison with Standard MCP Implementations

The current MCP implementations are based on the compliance with the protocol but might lack a comprehensive end-to-end demonstration of a frontend, a backend and real tools.

The major enhancements in this undertaking were:

- Added three tools (Filesystem, Browser, GitHub)
- Added a contemporary React frontend.
- Installed tool selection with the assistance of LLM.
- Added detailed tool call trace display.

Therefore, the project offers a more illustrative and comprehensive application of MCP principles. Moreover, the discussion of the tool servers integration shows how the MCP concepts can be scaled to the larger ecosystems and not be limited to the mere demonstrations. The project highlights how various instruments can work together by using a single-roof coordinator, context of which the system can undertake multi-stage tasks, which entails browsing, file manipulation as well as exploration of repositories jointly. This coordinated behaviour is more related to development situations in the real world where various tools are required. The final outcome is a

framework that does not only follow principles of protocol but also proves their worth in the form of interrelated functionality.

Collectively, these comparisons indicate that the suggested Unified MCP Framework can be considered more advanced in its ability and coverage in comparison to the current solutions. It gives a safe and transparent and modular environment that facilitates real tool execution as opposed to artificial behaviours. The system brings the AI-assisted development a step closer to the practical implementation as it allows agents to access files, browsers and repositories. These enhanced traceability, safety and extensibility make the framework suitable in the future investigations of agent-driven software engineering systems as protocol-based orchestration can contribute to higher productivity among developers substantially.

CHAPTER 6: CONCLUSIONS AND FUTURE SCOPE

6.1 CONCLUSION

This initiative resulted in creating a single platform for developing workflows using natural language based development processes through a combined MCP Framework consisting of the Host, Filesystem, Browser and GitHub MCP Servers along with a front end user interface. The structure provides easy, secure AI supported workflows by interpreting what the user wants to do, choosing the right tool to use, executing that tool in a manner that is safe, returning a structured output to the user that will serve as the foundation for agent-based automation. The central part of this technology is the Orchestration Layer, which allows users to operate all tools from a single interface, rather than having to switch between files, browsers or repositories. The Filesystem MCP Server allows for secure and sandboxed file operations, the Browser MCP Server allows for automated, stable and accurate text and image searching and the GitHub MCP Server allows for efficient retrieval of files and repository metadata. Additionally, through the front-end user interface, users have visibility into their queries, the decisions taken by the tools and the execution traces, thus making it much easier to debug and validate automation processes. As a whole, this initiative delivers modular, scalable, flexible, and responsive architecture to support the development and integration of workflows containing AI features.

6.2 FUTURE SCOPE

Even though the existing system manages to fulfill its primary goals, numerous opportunities remain that could be used to develop it and increase its functionality in the future. The following are the highlights of some of the avenues through which the project can be expanded both on the technical aspects and also with respect to usability.

- Increase of Tools -: In the current version, there is only a choice of tools which are integrated. The framework can be expanded to take into account enormous numbers of categories of tools in the future, including database query engines through which code can be executed directly on the data, remote execution modules through which code can be executed on an external machine, cloud-service utility e.g., storage or compute management and even email or alert-notification tools. New tools will be easy and consistent to add since all the tools adhere to the same MCP interface.

- Implementation of Full MCP Protocol -: The system is now based on the overall principles of the Model Context Protocol. Nevertheless, full official specification of MCP can be applied in subsequent versions. Making the entire protocol will enhance interoperability with other applications based on MCP and enable more advanced applications such as event streaming, richer tool metadata and allow more standardized communication between components.

- Developed Front-end Facilities -: Enhancing the features of the frontend to make it more user-friendly can also be an extensive improvement when a user has to use it over a long period or multiple steps. They are real-time execution history, multi-session chat history, role-based access controls and cleaner and more modern user interface design. This and similar enhancements will render the system more user-friendly to individuals and teams that require workflow sharing.

- Improvement of Performance and Efficiency -: The system can be optimally faster and more responsive in a number of ways. This can take the form of caching pages that are commonly visited, reusing browser context to avoid delay on startup and to optimize GitHub API calls to minimize latency and rate-limit problems. Through such enhancements , the overall execution time will go down and user experience will be made easier.

- Deployment and Packaging Options -: The whole framework is easily distributable. An alternative one is to package it as a program which can be easily installed by the end-users. The next alternative is that of containerizing the system with Docker enabling the system to run consistently on various systems. Also, the implementation of the framework on cloud-based platforms would enable the remote connection and make MCP system useable anywhere.

- Enhanced Protection of Security and Safety: Security will also become an even bigger concern as the system expands. The next iterations will consist of more detailed authorization options, less liberal sandboxing and a more detailed price ceiling of browser and API requests. These enhancements will safeguard the system against abuse, assure reliability and safe tools and external process execution.

REFERENCES

[1] F. Fiegel, “Introducing Model Context Protocol (MCP),” *Glama – MCP Hosting Platform*, 2024. https://glama.ai/blog/2024-11-25-model-context-protocol-quickstart (accessed Dec. 02, 2025).

[2] O. Shree, “Constrained Tool Design for LLMs: Model Context Protocol in Time Travel Debugging,” *Glama – MCP Hosting Platform*, 2025. https://glama.ai/blog/2025-10-27-agentic-debugging-with-time-travel-the-architecture-of-certainty (accessed Dec. 02, 2025).

[3] O. Shree, “Evaluating Tool-Oriented Architectures for AI Agents,” *Glama – MCP Hosting Platform*, 2025. https://glama.ai/blog/2025-09-02-comparing-mcp-vs-lang-chainre-act-for-chatbots (accessed Dec. 02, 2025).

[4] F. Fiegel, “Why AI Agents Need a New Protocol,” *Glama – MCP Hosting Platform*, 2025. https://glama.ai/blog/2025-06-06-mcp-vs-api (accessed Dec. 02, 2025).

[5] O. Shree, “Leveraging the MCP Registry for Structured Server Definition and Discovery,” *Glama – MCP Hosting Platform*, 2025. https://glama.ai/blog/2025-10-26-the-model-context-protocol-registry-standardizing-server-discovery-in-a-decentralized-ecosystem (accessed Dec. 02, 2025).

[6] O. Shree, “Secure Your Codebase: A Step-by-Step Guide to Automating PR Reviews,” *Glama – MCP Hosting Platform*, 2025. https://glama.ai/blog/2025-09-15-automating-git-hub-pull-request-security-checks-with-glama-ai-automation-feature (accessed Dec. 02, 2025).

[7] O. Shree, “How AI Agents Plan and Execute Commands on IoT Devices,” *Glama – MCP Hosting Platform*, 2025. https://glama.ai/blog/2025-08-22-agent-workflows-and-tool-design-for-edge-mcp-servers (accessed Dec. 02, 2025).

[8] O. Shree, “Securing Enterprise AI with an MCP Gateway,” *Glama – MCP Hosting Platform*, 2025. https://glama.ai/blog/2025-09-08-the-missing-gateway-centralizing-security-for-the-model-context-protocol (accessed Dec. 02, 2025).

[9] O. Shree, “Establishing First-Class Identity and Access Management for Agents using MCP,” *Glama – MCP Hosting Platform*, 2025. https://glama.ai/blog/2025-11-27-securing-enterprise-ai-agents-with-unique-identities-in-the-model-context-protocol-mcp (accessed Dec. 02, 2025).

[10] O. Shree, “Building Advanced AI Agents Using the Model Context Protocol (MCP) and mcp-agent,” *Glama – MCP Hosting Platform*, 2025. https://glama.ai/blog/2025-09-09-exposing-agents-as-mcp-servers-with-mcp-agent (accessed Dec. 02, 2025).

[11] H. Song *et al.*, “Beyond the Protocol: Unveiling Attack Vectors in the Model Context Protocol Ecosystem,” *arXiv.org*, 2025. https://arxiv.org/abs/2506.02040

[12] S. Fan, X. Ding, L. Zhang, and L. Mo, “MCPToolBench++: A Large Scale AI Agent Model Context Protocol MCP Tool Use Benchmark,” *arXiv.org*, 2025. https://arxiv.org/abs/2508.07575

[13] N. Croce and T. South, “Trivial Trojans: How Minimal MCP Servers Enable Cross-Tool Exfiltration of Sensitive Data,” *arXiv.org*, 2025. https://arxiv.org/abs/2507.19880

[14] M. M. Hasan, H. Li, E. Fallahzadeh, R. G. Krishnan, B. Adams, and A. E. Hassan, “Model Context Protocol (MCP) at First Glance: Studying the Security and Maintainability of MCP Servers,” *arXiv.org*, 2025. https://arxiv.org/abs/2506.13538

[15] N. V. Sai and I. Habler, “Enterprise-Grade Security for the Model Context Protocol (MCP): Frameworks and Mitigation Strategies,” *arXiv.org*, 2025. https://arxiv.org/abs/2504.08623

[16] N. Krishnan, “Advancing Multi-Agent Systems Through Model Context Protocol: Architecture, Implementation, and Applications,” *arXiv.org*, 2025. https://arxiv.org/abs/2504.21030

[17] J. Cui, “MCP Protocol’s Capability in Integrating LLM Models: A Study within High-Tech AIOps R&D Processes,” *ResearchGate*, May 17, 2025. https://www.researchgate.net/publication/391837315_MCP_Protocol

[18] H. Jing *et al.*, “MCIP: Protecting MCP Safety via Model Contextual Integrity Protocol,” *arXiv.org*, 2025. https://arxiv.org/abs/2505.14590

[19] A. Ehtesham, A. Singh, G. K. Gupta, and S. Kumar, “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.org*, 2025. https://arxiv.org/abs/2505.02279

[20] A.-A. Avram, A. Groza, and A. Lecu, “MCP-Orchestrated Multi-Agent System for Automated Disinformation Detection,” *arXiv.org*, 2025. https://arxiv.org/abs/2508.10143

[21] V. Ayyagari, “Model Context Protocol for Agentic AI: Enabling Contextual Interoperability Across Systems,” *International Journal of Computational and Experimental Science and Engineering*, vol. 11, no. 3, Aug. 2025, doi: https://doi.org/10.22399/ijcesen.3678.

[22] X. Hou, Y. Zhao, S. Wang, and H. Wang, “Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions,” *arXiv.org*, 2025. https://arxiv.org/abs/2503.23278

[23] S. Karimova, “he model context protocol: a standardization analysis for application integration,” *UNEC Journal of Computer Science and Digital Technologies*, vol. 2, no. 5, 2025, doi: https://doi.org/10.30546/uneccsdt.2025.01.050.

[24] H. Guo *et al.*, “A Measurement Study of Model Context Protocol,” *arXiv.org*, 2025. https://arxiv.org/abs/2509.25292

[25] Surya Rao Rayarao and Naga Donikena, “Bridging AI and External Systems: A Comprehensive Analysis of the Model Context Protocol (MCP),” *Authorea*, Aug. 2025, doi: https://doi.org/10.22541/au.175555093.31837381/v1.

[26] C. Avila, D. Ilbay, and D. Rivera, “Human–AI Teaming in Structural Analysis: A Model Context Protocol Approach for Explainable and Accurate Generative AI,” *Buildings*, vol. 15, no. 17, p. 3190, Sep. 2025, doi: https://doi.org/10.3390/buildings15173190.

[27] H. Li, Y. Xu, and T. Hong, “EnergyPlus-MCP: A model-context-protocol server for ai-driven building energy modeling,” *SoftwareX*, vol. 32, pp. 102367–102367, Sep. 2025, doi: https://doi.org/10.1016/j.softx.2025.102367.

8% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

Match Groups

63 Not Cited or Quoted 7%
Matches with neither in-text citation nor quotation marks

0 Missing Quotations 0%
Matches that are still very similar to source material

7 Missing Citation 1%
Matches that have quotation marks, but no in-text citation

0 Cited and Quoted 0%
Matches with in-text citation present, but no quotation marks

Top Sources

8% Internet sources
3% Publications
5% Submitted works (Student Papers)

Integrity Flags

1 Integrity Flag for Review

Replaced Characters
421 suspect characters on 6 pages
Letters are swapped with similar characters from another alphabet.

Our system's algorithms look deeply at a document for any inconsistencies that would set it apart from a normal submission. If we notice something strange, we flag it for you to review.

A Flag is not necessarily an indicator of a problem. However, we'd recommend you focus your attention there for further review.

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT

PLAGIARISM VERIFICATION REPORT

Date: 1/12/25

Type of Document (Tick): PhD Thesis | M.Tech/M.Sc. Dissertation | B.Tech./BCA/BBA Report ✓

Name: Anjali, Omshree, Harsh Naviya Department: CSE Enrolment No 22103040, 22103041, 22103050, 22103113

ORCID ID. SCOPUS ID.

Contact No. 9027063789 E-mail. Anjali : 22103113 @jiit.solan.in

Name of the Supervisor: Dr Aman sharma

Title of the Thesis/Dissertation/Project Report/Paper (In Capital letters): UNIFIED MCP FRAMEWORK FOR CONTEXT-AWARE AI AGENTS IN SOFTWARE DEVELOPMENT.

UNDERTAKING

I undertake that I am aware of the plagiarism related norms/ regulations, if I found guilty of any plagiarism and copyright violations in the above thesis/report even after award of degree, the University reserves the rights to withdraw/revoke my degree/report. Kindly allow me to avail Plagiarism verification report for the document mentioned above.

- Total No. of Pages = 60
- Total No. of Preliminary pages = 9
- Total No. of pages accommodate bibliography/references = 2

Omshree Naviya Anjali Harsh

(Signature of Student)

FOR DEPARTMENT USE

We have checked the thesis/report as per norms and found **Similarity Index :** 8 (%) and **AI Writing: 0%** [✱] or *% [ ]. (Please [✓] any one % as per generated report). Therefore, we are forwarding the complete thesis/report for final plagiarism check. The plagiarism verification report may be handed over to the candidate.

Aman 2/12/25

(Signature of Guide/Supervisor) Signature of HOD

FOR LRC USE

The above document was scanned for plagiarism check. The outcome of the same is reported below:

Document Received Date	Excluded	Similarity Index (%)		Title, Abstract & Chapters Details	
	All Preliminary Pages	Overall Similarity		Word Counts	
Report Generated Date	Bibliography / References	AI Writing		Character Counts	
	Images/Quotes	0%		Page counts	
	14 Words String	*%		File Size	

Checked by
Name & Signature Librarian

Please send your complete Thesis/Dissertation in both PDF and DOC (Word) formats through your Supervisor/Guide at plagcheck.juit@gmail.com

(Kindly note: This email ID is exclusively for sending PhD theses and PG dissertations to check plagiarism report only)