UDC 004.8:004.738.5:025.4.03
DOI https://doi.org/10.36994/2788-5518-2025-02-10-03

ARCHITECTURE AND IMPLEMENTATION OF A MODEL CONTEXT PROTOCOL-BASED DOCUMENTATION ASSISTANT: A CASE STUDY WITH AI-POWERED CONTEXT GENERATION

***Fant M. O.**, PhD, Zhytomyr Polytechnic State University, Zhytomyr, Ukraine. https://orcid.org/0000-0002-4994-8009, fantkolja@gmail.com*

***Vakaliuk T. A.**, D.Sc., Prof., Zhytomyr Polytechnic State University, Zhytomyr, Ukraine. https://orcid.org/0000-0001-6825-4697, tetianavakaliuk@gmail.com*

***Abstract.** Modern technical documentation systems face increasing complexity and volume, making it difficult for users to obtain accurate, contextually relevant answers without substantial manual navigation. Traditional keyword-based search mechanisms remain limited because they rely on lexical matching rather than understanding user intent or synthesising information spread across multiple documentation sources. As a result, users often receive fragmented results and must piece together separate sections to find complete answers.*

This article presents the architecture, design principles, and implementation details of an intelligent documentation assistant based on the Model Context Protocol (MCP). MCP provides a standardised integration mechanism that enables seamless communication between documentation sources and Large Language Models (LLMs). Our proposed system extends existing documentation workflows by incorporating AI-powered context generation capable of automatically analysing, chunking, and structuring raw documentation into MCP-compatible resources. This approach ensures that LLM responses remain grounded in authoritative, up-to-date documentation and reduces the risk of hallucinations.

The architecture consists of four interconnected layers: the documentation website, the MCP server, the AI-powered context generation pipeline, and the LLM integration layer. Each layer is responsible for a distinct set of functions, enabling a clean separation of concerns and improving maintainability. Through a detailed case study, we demonstrate how the system dynamically selects relevant documentation chunks, retrieves context on demand, and produces semantically accurate answers.

Experimental results show that the MCP-based assistant significantly improves response quality compared to traditional search systems, offering better semantic relevance, higher completeness, and substantial reductions in hallucination rates. The standardised MCP ecosystem also enhances interoperability, scalability, and ease of deployment. The findings highlight MCP's potential as a foundational protocol for future intelligent documentation systems.

***Key words:** model context protocol, large language models, AI architecture, context generation, retrieval-augmented generation.*

АРХІТЕКТУРА ТА РЕАЛІЗАЦІЯ АСИСТЕНТА ДОКУМЕНТАЦІЇ НА ОСНОВІ ПРОТОКОЛУ КОНТЕКСТУ МОДЕЛІ: ПРИКЛАД ГЕНЕРАЦІЇ КОНТЕКСТУ ЗА ДОПОМОГОЮ ШТУЧНОГО ІНТЕЛЕКТУ

***Микола Фант**, к.ф.н., Державний університет «Житомирська політехніка», Житомир, Україна. https://orcid.org/0000-0002-4994-8009, fantkolja@gmail.com*

***Тетяна Вакалюк**, д.п.н., проф., Державний університет «Житомирська політехніка», Житомир, Україна. https://orcid.org/0000-0001-6825-4697, tetianavakaliuk@gmail.com*

***Анотація.** Сучасні системи технічної документації постійно зростають за обсягом і складністю, що ускладнює для користувачів отримання точних та контекстно релевантних відповідей без значних зусиль з навігації. Традиційні механізми пошуку, побудовані на ключових словах, мають суттєві обмеження, оскільки спираються на лексичну відповідність, а не на розуміння намірів користувача чи здатність синтезувати інформацію, розподілену між різними джерелами документації. У результаті користувачі стикаються з фрагментованими відповідями та змушені самостійно об'єднувати окремі частини, щоб отримати повну інформацію.*

У статті представлено архітектуру, концептуальні принципи та особливості реалізації інтелектуального асистента документації, побудованого на основі протоколу контексту моделі (MCP). MCP забезпечує стандартизований спосіб інтеграції між джерелами документації та великими мовними моделями. Запропоноване рішення доповнює традиційні підходи за допомогою генерації контексту на основі ШІ, яка автоматично аналізує, сегментує та структурує сирий текст

документації у ресурси, сумісні з MCP. Такий підхід гарантує, що відповіді LLM залишаються точними, заснованими на актуальних даних і менш схильними до галюцинацій.

Архітектура складається з чотирьох взаємопов'язаних рівнів: вебсайту документації, сервера MCP, системи ШІ-генерації контексту та шару інтеграції LLM. Кожен рівень виконує чітко визначені функції, що підвищує масштабованість і спрощує підтримку системи. На основі детального прикладу продемонстровано, як система автоматично визначає релевантні фрагменти документації, отримує необхідний контекст і формує семантично точні відповіді.

Результати дослідження показують, що асистент на основі MCP суттєво підвищує якість відповідей порівняно з традиційним пошуком, забезпечує вищу семантичну релевантність, повноту інформації та значне зменшення рівня галюцинацій. Стандартизована інфраструктура MCP також покращує сумісність, гнучкість і простоту розгортання. Отримані результати підтверджують потенціал MCP як основи для майбутніх інтелектуальних систем документації.

Ключові слова: *протокол контексту моделі, велика мовна модель, архітектура штучного інтелекту, генерація контексту, генерація з доповненим пошуком.*

Introduction

Technical documentation has become increasingly complex and voluminous as software systems grow in sophistication. Traditional keyword-based search systems suffer from fundamental limitations in understanding user intent and context, requiring users to formulate precise queries using exact terminology found in the documentation [1]. Modern documentation often spans multiple interconnected topics, requiring users to synthesise information from various sources to answer a single question, yet traditional search returns isolated pages or sections, leaving the integration work to the user.

The emergence of LLMs presents a potential solution through their ability to understand natural language queries and synthesise information [2]. However, deploying LLMs for documentation assistance introduces new problems. LLMs are prone to "hallucinations" – generating plausible but incorrect information – particularly when asked about specific technical details not present in their training data [3; 4]. Additionally, LLMs have knowledge cutoff dates, making them unsuitable for answering questions about current software versions or recent features without access to up-to-date documentation [5; 6].

Custom implementations of LLM-documentation integration often result in tightly coupled architectures that are difficult to maintain or migrate. There is a clear need for a standardised protocol that enables seamless integration between documentation resources and LLM systems while ensuring responses are grounded in current, accurate documentation [7]. The MCP addresses this standardisation gap by providing a universal protocol for connecting LLM applications with data sources. MCP defines a precise specification for how context providers can expose their resources, tools, and prompts to LLM clients in a consistent, discoverable manner [8]. This protocol-based approach offers several advantages such as implementation complexity reduction, interoperability between different systems, and clear separation of concerns [9].

The emergence of AI-powered context generation tools, such as KapaAI, further motivates this research by offering automated solutions for extracting and structuring content from documentation. These tools can analyse documentation, identify semantic relationships, create appropriate chunking strategies, and generate structured contexts optimised for LLM consumption. The combination of MCP's standardised protocol and AI-powered context generation creates an opportunity to build documentation assistants that are both powerful and maintainable.

From a practical perspective, organisations invest significant resources in creating and maintaining technical documentation, yet often struggle to ensure users can effectively access this information. An intelligent documentation assistant would substantially improve the return on investment in documentation, reduce the support burden, accelerate developer onboarding, and enhance overall documentation usability.

The primary objective of this research is to design and implement an MCP-based architecture for an intelligent documentation assistant that delivers accurate, contextually relevant answers to user queries.

Results. The proposed intelligent documentation assistant employs a four-layer architecture designed around the MCP as the central integration mechanism. This architecture separates concerns while enabling seamless communication between components, creating a maintainable and scalable system. The architecture follows a user-initiated request-response pattern. When a user submits a natural language query, the request flows through each layer as shown in figure 1.

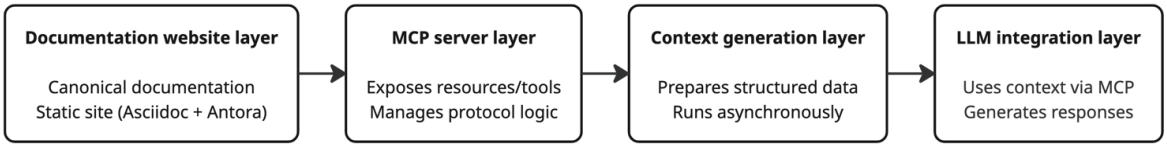

Fig. 1. Four-layer architecture flow

A key architectural principle is the separation of documentation management from intelligence capabilities. The documentation system continues to function as a traditional website, serving human readers through conventional interfaces. The MCP layer adds LLM accessibility without requiring modifications to the core documentation structure, ensuring that improvements to either component can proceed independently.

The component interaction model is built on MCP's client-server paradigm. The LLM system acts as an MCP client, discovering and accessing resources exposed by the MCP server. The server draws from pre-processed contexts generated by the AI-powered context generation system. This model provides flexibility – the context generation process can run asynchronously, updating MCP resources without interrupting active queries.

The Documentation website layer hosts the authoritative documentation content. In our implementation, this layer utilises Netlify as the hosting platform, chosen for its support for static site generation, continuous deployment, and edge network distribution. The documentation is written in AsciiDoc and built with Antora.

The website's structure adheres to standard documentation patterns, featuring clear navigation hierarchies and consistent formatting. Each page includes metadata such as title, description, and tags, which become valuable inputs for context generation. The documentation follows a 'docs-as-code' approach, where updates flow through the same review processes as code changes. The build process transforms asciidoc into static HTML and triggers the context generation pipeline, ensuring MCP resources remain synchronised.

Notably, the documentation layer remains unaware of the MCP infrastructure. No special markup is required in the documentation to support the intelligent assistant, ensuring maintenance can be performed using standard tools and workflows.

The MCP server layer implements the MCP specification, exposing documentation resources and tools to LLM clients. The server translates between the MCP protocol and the underlying documentation system, implementing the full MCP lifecycle: initialisation, resource discovery, resource access, tool invocation, and session management.

The server exposes documentation through MCP resources, which represent addressable pieces of content. Each resource corresponds to a semantically meaningful chunk – a complete page, specific section, or conceptual unit. Resources include metadata such as MIME type, URI, and annotations that help the LLM understand the resource's purpose. Resources are organised hierarchically, mirroring the documentation's logical structure.

The server also implements tools that enable interaction beyond simple retrieval. A search tool allows querying documentation using keywords or phrases, returning ranked resource identifiers. A navigation tool helps understand the documentation structure and locate related content. These tools transform documentation from a passive resource into an active, queryable knowledge base.

The implementation uses Node.js and the official MCP SDK, ensuring protocol compliance. The server maintains an in-memory index for fast discovery and implements caching to minimise redundant processing. For Netlify deployment, the server can be implemented as serverless functions or edge functions. Security mechanisms validate authorised clients, implement rate limiting, and log all access patterns.

The Context generation layer transforms raw documentation into structured, semantically optimised contexts suitable for LLM consumption. This layer employs KapaAI to automate documentation analysis, chunking, and context preparation [10]. The goal is to create contexts that maximise the LLM's ability to provide accurate answers while fitting within context window constraints.

The process begins with the ingestion of documentation, which involves reading files from the build output or source repository. It processes various content types, including markdown, HTML, and code examples, extracting metadata such as titles, headings, and cross-references.

Unlike naive approaches that split at arbitrary boundaries, semantic chunking identifies logical information units that can stand alone. KapaAI uses natural language understanding to identify semantic boundaries, ensuring chunks maintain coherence. A chunk might be a complete concept explanation, a tutorial step, or an API endpoint description with parameters and examples.

The chunking strategy balances competing goals, since chunks must be small enough to fit within context windows yet large enough to contain sufficient information. Overlapping strategies ensure that information spanning boundaries remains accessible and transparent. The system maintains metadata linking related chunks, enabling the LLM to request additional context when needed.

Each chunk is enriched with metadata, including the source documentation page, position within the hierarchy, related topics, code language, and semantic tags. This metadata becomes part of the MCP resource description, allowing informed decisions about which resources to access. The system also creates relationships between chunks, identifying prerequisites, related topics, and examples.

The output is a structured JSON dataset describing all chunks, their content, metadata, relationships, and organisation. This dataset serves as the foundation for the MCP server's resource index. The generation process is incremental, regenerating only chunks affected by documentation updates.

The LLM integration layer connects user queries to the MCP-based documentation system and generates natural language responses based on retrieved context. This layer implements the MCP client protocol, enabling it to discover and access documentation resources. The integration is designed to be LLM-agnostic, supporting various models including Sonnet 4.5, GPT-5, and others.

When a user submits a question, the system analyses the query to understand intent and identify relevant documentation topics. This analysis determines which MCP resources and tools to invoke. Prompt engineering is central to effectiveness – the system constructs prompts including the query, relevant documentation context, and instructions emphasising accuracy and citation. The prompts specify response format based on query characteristics.

Since context window management is critical, the integration layer implements strategies to maximise effective context utilisation, retrieving only the most relevant chunks, prioritising recent or frequently accessed content, and truncating peripheral information when necessary. The system maintains conversation history for multi-turn interactions, enabling coherent conversations where the LLM can reference previous answers.

Response generation includes citation and attribution. When answers are based on specific documentation sections, responses include references or links. The implementation extracts source attribution from MCP resource metadata, formatting citations naturally. Error handling ensures robust operation – if the MCP server is unavailable, the system might fall back to cached contexts or inform the user of reduced capability. Table 1 summarises the implementation details, key technologies, responsibilities, and integration aspects of each major layer in the system architecture.

Table 1
Layers description

Layer	Implementation details	Technologies	Responsibilities	Integration aspect
Documentation website	Built with Antora + Asciidoc; hosted on Netlify	Antora, Asciidoc, Netlify	Hosts canonical content; triggers context updates	Independent of MCP
MCP server	Node.js MCP SDK, serverless deployment	Node.js, MCP SDK	Exposes resources, tools, handles search, and navigation	Translates between docs and LLM
Context generation	Automated semantic chunking with KapaAI	KapaAI, JSON-RPC	Generates structured contexts & metadata	Feeds the MCP resource index
LLM integration	MCP client + prompt engineering	LLM, AI-agent	Handles query analysis, context selection, and response generation	Consumes MCP resources

The technology stack strikes a balance between protocol compatibility, deployment flexibility, performance, and ecosystem maturity. The documentation website layer employs a static site generator (Antora + Asciidoc) that transforms .adoc files into optimised HTML. Netlify serves as the hosting platform, providing global CDN distribution, automatic HTTPS, and serverless function hosting.

The MCP server layer is implemented in Node.js, running on Netlify Edge Functions, using the official MCP SDK from Anthropic. Node.js offers excellent performance for I/O-bound operations and straightforward deployment to serverless platforms. The MCP SDK provides TypeScript types and utilities ensuring protocol compliance. Netlify Edge Functions enable low-latency execution by running serverless code at the network edge, allowing the MCP server to deliver documentation resources closer to end users for improved responsiveness.

For the context generation layer, KapaAI serves as the AI-powered processing engine, providing APIs for documentation ingestion, semantic analysis, and chunking optimised for LLM applications. Integration occurs through its REST API, with data transformation between KapaAI's output and the MCP resource schema.

The implementation of LLM integration layer utilises MCP's official SDK, which includes MCP client capabilities. Development tools include Git, GitHub Actions for CI/CD, and monitoring tools like Grafana.

The server implementation provides a streamlined integration layer that exposes Hazelcast documentation retrieval capabilities to LLM systems through the Model Context Protocol. Deployed as a Netlify Edge Function, this approach leverages edge computing infrastructure to provide low-latency access to documentation context with minimal operational overhead [10].

Rather than implementing the full MCP specification from scratch, the server utilises the official MCP SDK (@modelcontextprotocol/sdk) combined with the Netlify adapter (@modelfetch/netlify) to handle the complexities of JSON-RPC communication and Server-Sent Events (SSE) streaming in edge environments [11]. This architectural decision addresses the unique constraints of edge computing platforms, where traditional Node.js HTTP handling mechanisms are unavailable.

The @modelfetch/netlify adapter implements the streamingHttp protocol, which internally uses StreamableHTTPServerTransport to manage SSE streams in serverless edge contexts. This adapter layer

performs several critical functions: it translates edge fetch Request / Response objects to Node-style HTTP transports expected by the MCP SDK, parses incoming JSON-RPC payloads, routes protocol methods (initialise, tool:discover, tool:invoke) to registered tools, manages SSE streaming for real-time responses, and handles error formatting according to JSON-RPC specifications [12].

Server initialisation loads configuration from environment variables, including the KapaAI API key, project ID, and integration ID. These credentials enable authenticated access to the pre-processed documentation contexts that KapaAI has generated from the Hazelcast documentation site. The server is configured with metadata identifying it as "Hazelcast Docs MCP" with version tracking for compatibility management:

```
const server = new McpServer({
  name: 'Hazelcast Docs MCP',
  version: SERVER_VERSION,
})
```

The server exposes a single tool called "ask_hazelcast_docs" that serves as the primary interface for documentation queries. This tool is registered with comprehensive metadata that helps LLM clients understand when and how to invoke it.

The implementation includes sophisticated request handling logic to distinguish between browser requests and MCP client requests. When browser traffic is detected (through User-Agent headers and Accept headers indicating text / html), the server performs a redirect to the documentation homepage rather than returning raw JSON-RPC responses. This ensures a better user experience when the endpoint is accidentally accessed through a web browser.

For legitimate MCP client requests, the server enforces protocol requirements by patching request headers to include both application/json and text/event-stream in the Accept header, as required by the MCP specification. The server also enforces the POST method for the /mcp endpoint, returning a 405 Method Not Allowed response for other HTTP methods.

The Edge Function configuration includes rate limiting to protect against abuse and prevent unexpected usage spikes that could lead to cost overruns. The rate limit is set to 20 requests per 120-second window, aggregated by both IP address and domain. This balanced approach allows legitimate usage while preventing denial-of-service scenarios and controlling API costs:

```
export const config = {
  path: '/mcp',
  rateLimit: {
    windowLimit: 20,
    windowSize: 120,
    aggregateBy: ["ip", "domain"],
  }
}
```

To facilitate seamless interaction with the MCP server, users can connect using either the MCP Inspector tool or directly from Visual Studio Code. Below are brief guides for each method, with placeholders for corresponding figures.

To initiate a connection from the MCP Inspector, open the tool and navigate to the connection settings panel (see figure 2). Enter the server's endpoint URL, ensuring you use the correct protocol (such as HTTPS) and port. Then, click "Connect" to establish communication. Once connected, you can inspect requests and responses in real time.

Within Visual Studio Code, install the MCP extension if it is not already present. Then, open the command palette and select "MCP: Connect to Server" (see figure 3). Enter the server's endpoint when prompted, and the extension will handle authentication and establish the connection. A successful connection will enable you to interact directly with the server from your code editor.

Implementation and evaluation revealed several key findings about the MCP-based architecture and its effectiveness in providing documentation assistance. The architecture successfully decouples documentation management from AI capabilities, allowing for the independent evolution of each component. Documentation teams can update content without affecting the assistant, and improvements to the LLM integration do not require changes to the documentation.

AI-powered context generation through KapaAI proved highly effective at creating semantically coherent chunks. AI-generated chunks preserved conceptual boundaries better than rule-based approaches, recognising that code examples and explanations form units that should remain together. The impact of the key findings is shown in Table 2.

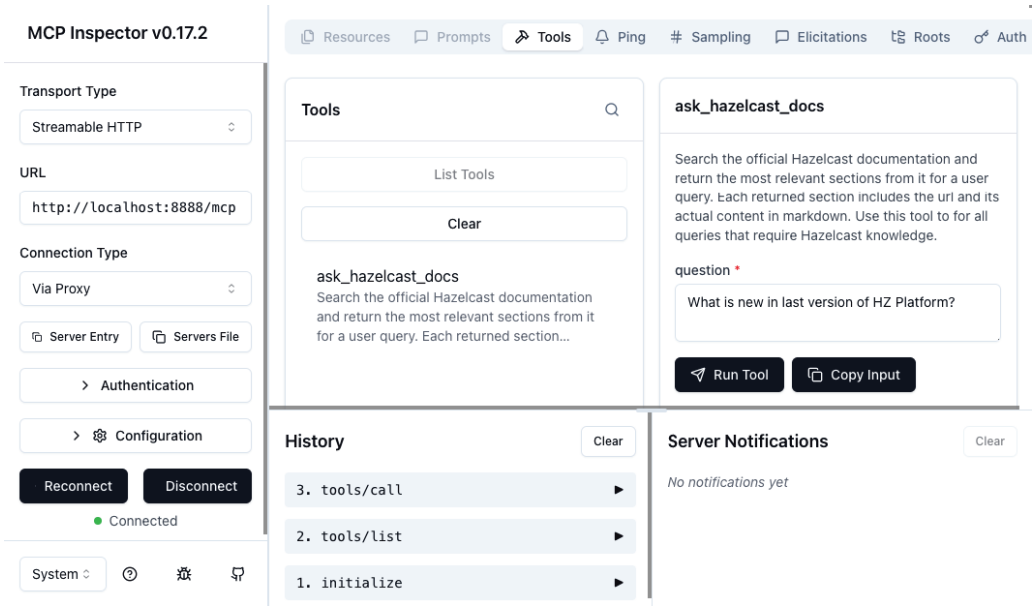

Fig. 2. Connecting from the MCP Inspector

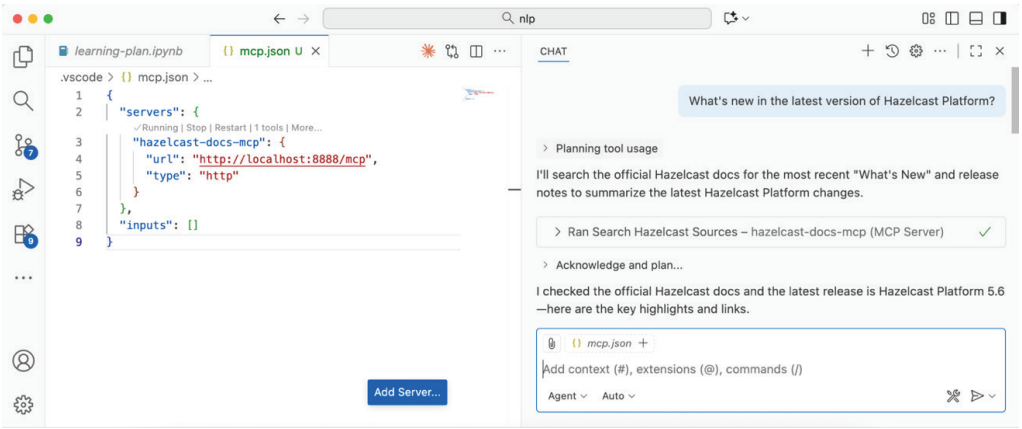

Fig. 3. Connecting from the MCP Inspector

Table 2
Key findings

Aspect	Finding	Impact
Hallucination Reduction	Reduction vs. ungrounded LLM	Significantly improved answer accuracy
Optimal Context Size	5,000-10,000 tokens	Best balance of relevance and performance
Answer Completeness	Multi-source synthesis	Better than traditional keyword search

Context window management emerged as critical. Even with large context windows, careful chunk selection is essential – including too much context can degrade response quality by diluting attention. The optimal strategy retrieves only the most relevant chunks rather than maximising context usage.

The layered architecture, with clear separation of concerns, proved highly maintainable, as each layer was developed and tested independently. The REST API wrapper approach (rather than full MCP implementation) provided the necessary functionality without excessive complexity, demonstrating that pragmatic solutions often outperform over-engineered ones.

Conclusions. This article presented the architecture and implementation of an intelligent documentation assistant built on the MCP, demonstrating a practical approach to integrating LLM with technical documentation systems. The proposed four-layer architecture – comprising a documentation website, an MCP server, AI-powered context generation, and LLM integration – successfully separates concerns while enabling seamless information flow from documentation sources to user responses.

Implementation findings validate the architectural approach while revealing important practical considerations. The MCP protocol introduces minimal overhead while providing significant benefits in

standardisation, maintainability, and flexibility. Response quality improvements over both traditional search and ungrounded LLM queries are substantial, with marked reductions in hallucination rates and increases in answer completeness. However, success depends critically on the quality of documentation – the intelligent assistant amplifies both the strengths and weaknesses of existing documentation.

The system's practical implications extend beyond immediate documentation assistance. The patterns demonstrated – utilising MCP as an integration layer, employing AI for automated context preparation, and combining retrieval with generation – apply to a wide range of knowledge-intensive applications. Organisations implementing similar systems should anticipate shifts in documentation workflows, support patterns, and user behaviour.

The MCP represents an important step toward standardised LLM integration patterns. By adopting and contributing to such open protocols, the technical community can avoid fragmentation and proprietary lock-in while enabling innovation and interoperability. The documentation assistant use case serves as a compelling demonstration of MCP's value and a template for future applications, successfully achieving the research objectives of designing an effective MCP-based system, investigating integration patterns, evaluating AI-powered context generation, and providing practical guidance for practitioners.

References

1. Huang L., Yu W., Ma W., Zhong W., Feng Z., Wang H., Chen Q., Peng W., Feng X., Qin B., Liu T. (2023). A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. arXiv preprint arXiv:2311.05232.

2. Gao Y., Xiong Y., Gao X., Jia K., Pan J., Bi Y., Dai Y., Sun J., Wang M., Wang H. (2023). Retrieval-augmented generation for large language models: A survey. arXiv preprint arXiv:2312.10997.

3. Xu Z., Jain S., Kankanhalli M. (2024). Hallucination is inevitable: An innate limitation of large language models. arXiv preprint arXiv:2401.11817.

4. Tonmoy S. M. T. I., Zaman S. M., Jain V., Rani A., Rawte V., Chadha A., Das A. (2024). A comprehensive survey of hallucination mitigation techniques in large language models. arXiv preprint arXiv:2401.01313.

5. Cheng J., Liu Q., Liu T., Wang J., Zhou A., Sun X. (2024). Dated data: Tracing knowledge cutoffs in large language models. arXiv preprint arXiv:2403.12958.

6. Kasai J., Sakaguchi K., Takemoto Y., Lee K., Bisk Y. (2023). Evaluating GPT-3.5 and GPT-4 on diverse factual text generation tasks. arXiv preprint arXiv:2305.14251.

7. Anthropic. (2024). Model Context Protocol specification. URL: https://modelcontextprotocol.io/

8. Lewis P., Perez E., Piktus A., Petroni F., Karpukhin V., Goyal N., Küttler H., Lewis M., Yih W., Rocktäschel T., Riedel S., Kiela D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. In Advances in Neural Information Processing Systems. 33. 9459–9474.

9. Ram O., Levine Y., Dalmedigos I., Muhlgay D., Shashua A., Leyton-Brown K., Shoham Y. (2023). In-context retrieval-augmented language models. T*ransactions of the Association for Computational Linguistics*. 11. 1316–1331.

10. kapa.ai. (2025, June 27). Build an MCP Server with kapa.ai. *kapa.ai Blog*. URL: https://www.kapa.ai/blog/build-an-mcp-server-with-kapa-ai

11. Netlify. (2024). Writing MCPs on Netlify: A guide to implementing Model Context Protocol servers on Netlify Edge Functions. *Netlify Developer Documentation*. URL: https://developers.netlify.com/guides/write-mcps-on-netlify/

12. Model Context Protocol. (2024). @modelfetch/netlify: Netlify adapter for Model Context Protocol servers (Version 1.0.0) [Computer software]. *npm*. URL: https://www.npmjs.com/package/@modelfetch/netlify

Дата першого надходження статті до видання: 23.10.2025
Дата прийняття статті до друку після рецензування: 25.11.2025
Дата публікації (оприлюднення) статті: 26.12.2025