Accepted for MODELS 2026. This is the camera-ready author version of the paper, the final version is accessible via the ACM digital library at: https://doi.org/10.1145/3822455.3830317.

Realizing a Model Context Protocol Server for the Graphical Language Server Platform

Andreas Hell
Business Informatics Group, TU Wien
Vienna, Austria
e12123458@student.tuwien.ac.at

Martin Fleck
EclipseSource Services GmbH
Vienna, Austria
mfleck@eclipsesource.com

Philip Langer
EclipseSource Services GmbH
Vienna, Austria
planger@eclipsesource.com

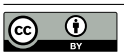
Dominik Bork
Business Informatics Group, TU Wien
Vienna, Austria
dominik.bork@tuwien.ac.at

Abstract

Model-Driven Engineering (MDE) offers strong abstractions but remains hindered in practice by complex tooling and steep learning curves. Recent advances in Large Language Models (LLMs) promise intuitive, natural-language interaction with modeling environments; however, existing approaches—such as direct API interaction, DSL generation, or manipulation of serialized models—exhibit limited reliability, scalability, and semantic precision. This paper presents a structured integration of LLMs into graphical modeling environments via the Model Context Protocol (MCP). We design and implement a reusable MCP server for the Graphical Language Server Platform (GLSP), enabling LLM agents to interact with models through semantically grounded, tool-based operations rather than direct model generation. We contribute (i) a reference architecture for MCP-based integration into GLSP derived from a systematic literature review of MCP implementations, (ii) design principles for mapping modeling concepts to MCP primitives and abstraction levels, and (iii) an empirical evaluation assessing the impact of MCP-based interaction on downstream modeling tasks. Our results show that MCP-based interactions significantly improve both information retrieval (perfect precision/recall across tasks) and model manipulation (reducing syntactic, semantic, and pragmatic errors), while also improving scalability and efficiency compared to serialization-based approaches. The MCP server enables robust modification of existing models, a key limitation of prior approaches. These findings demonstrate that structured, tool-mediated interaction is a crucial enabler for reliable AI-assisted modeling. The presented MCP–GLSP Server constitutes a reusable artifact applicable across GLSP-based tools, providing a practical foundation to advance intelligent modeling assistance.

CCS Concepts

• **Software and its engineering → Development frameworks and environments**; • **Computing methodologies → Artificial intelligence**.



MODELS 2026, Málaga, Spain

ACM ISBN 979-8-4007-2809-9/2026/10
https://doi.org/10.1145/3822455.3830317

Keywords

Model Context Protocol, MCP, Language Server Protocol, LSP, Intelligent Modeling Assistance, Large Language Models, LLM, GLSP

ACM Reference Format:
Andreas Hell, Martin Fleck, Philip Langer, and Dominik Bork. 2026. Realizing a Model Context Protocol Server for the Graphical Language Server Platform. In *ACM/IEEE 29th International Conference on Model Driven Engineering Languages and Systems (MODELS 2026), October 04–09, 2026, Málaga, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3822455.3830317

1 Introduction

Modeling has been part of the software engineering and architecture ecosystem for a long time, whether for creating supporting documents or as primary artifacts in Model-Driven Engineering (MDE). However, industry adoption, especially of full-fledged MDE, has been challenging due to cumbersome tooling coupled with steep learning curves [17, 29]. This has led to a varied landscape, driven by engineers seeking alternatives. Some resort to simple diagramming tools (e.g., Microsoft Visio), which sacrifice semantics but are consequently easier to use and allow full visual control [17]. Others use textual Domain-Specific Languages (DSLs) like PlantUML for efficiency, trading visual control for an intuitive textual interface, a flat learning curve, and easier version control, among other benefits [36]. In parallel to these 'workarounds', research and tool vendors embarked on the promising direction of Machine Learning and AI for Model-driven engineering (ML4MDE) [6, 22, 33, 37], and, more recently, on Large Language Models and MDE (LLM4MDE) [9].

The integration of LLMs into graphical modeling tools offers the opportunity to solve this problem at its root. By providing an easy-to-use textual interface, it enables intuitive and efficient modeling assistance [30], while retaining rich semantics and full visual control. However, current LLM4MDE approaches are insufficient for reliable, high-precision modeling. Specifically, two primary problems currently persist:

- **Unstructured API Interaction:** Directly employing LLMs to interact with modeling tool APIs in an unstructured manner leads to low reliability and erroneous tool invocation [31].
- **Semantic and Syntactic Information Representation:** LLMs are insufficient when working directly on a serialized model (e.g., the model's XML representation) [28]; they

should rather use a representation that helps mitigate the risk of token excess, syntactic errors, and semantic errors [1, 11].

Consequently, there is a lack of a structured protocol that enables an LLM to serve as a proper modeling assistant in graphical environments. Without such an integration, LLM-assisted modeling risks remaining limited by poor precision and recall in information retrieval and by errors during model creation/completion as models grow in size. This problem is measurable by the error rates and retrieval accuracy observed when LLMs work directly with serialized models or DSLs; the extent to which this will be explored is part of the evaluation in this paper.

To overcome these limitations, we propose using the Model Context Protocol (MCP), which provides a standardized mechanism for exposing tool capabilities and contextual information to LLMs, enabling more reliable, scalable, and semantically grounded interaction [34]. This paper investigates whether structured MCP-based interaction improves downstream modeling tasks compared to serialization-based approaches. Specifically, we develop an MCP server and demonstrate its integration into the client-server architecture of the Graphical Language Server Platform (GLSP) [27]. The primary goal of this MCP-GLSP server is to improve LLM-based modeling performance over the alternative LLM interaction methods. Our solution enables tool developers to easily build their intelligent GLSP modeling assistants on top of our MCP-GLSP integration. To realize such an integration, we report on i) the analysis of the conceptual and technical foundations required to integrate MCP into GLSP; ii) the analysis of the conceptual and technical foundations required to integrate MCP into GLSP; and iii) the design and implementation of a functional MCP-GLSP server capable of exposing model context and operations to an LLM client while offering model representations that reduce token usage without sacrificing syntactic and semantic information.

The quality of our solution is defined by its ability to outperform alternative methods across five key metrics, conceptually separated into the categories of *model reading* and *model construction*:

- **Model Reading (Information Retrieval):** Higher *Precision* and *Recall* when extracting information from the model, showing that this interaction model is superior in extracting correct and complete information.
- **Writing (Model Manipulation):** A significant reduction in *Syntax Errors*, *Semantic Errors*, and *Pragmatic Errors*, meaning that the resulting model not only adheres to a given language's specification, but also to the intentions of the user.

The remainder of this paper is structured as follows: Section 2 defines the relevant theoretical and conceptual background by introducing the Model Context Protocol and other relevant protocols. A systematic literature review of MCP implementations is presented in Section 3, which informs the conceptual design and implementation decisions for the MCP-GLSP integration discussed in Section 4. The results of a systematic evaluation and the experiences of integrating the MCP-GLSP solution into existing modeling editors are discussed in Section 5 before this paper is concluded in Section 6.

2 Background

This section provides a brief overview of the areas directly related to this research. Note that technologies such as the Graphical Language Server Platform (GLSP) and Large Language Models (LLMs) are widely reported in recent publications in the community. We therefore briefly introduce only the most relevant parts specific to our solution.

2.1 Graphical Language Server Platform

The **Graphical Language Server Platform (GLSP)** is an open-source project of the Eclipse Foundation maintained by EclipseSource. It is largely driven by industrial usage requirements, while there are also academic applications [19, 26]. GLSP *"is a client-server framework for building web-based diagram editors. It follows an architectural pattern similar to the hugely popular Language Server Protocol, but applies it to graphical modeling and diagram editors."* [12] An important aspect of our MCP-GLSP server is how behavior, such as a modeler interacting with GLSP-based tools, is handled. The GLSP client and server each have an action dispatcher that determines whether a given event should propagate through the internal event flow or be sent to its counterpart (cf. Fig. 1). Since the server is the passive partner in this process, all event flows start from the client, triggered by events such as the client's startup or the dragging of an element. Any user interaction (or tool interaction) is sent as an action to the GLSP server, which, upon receipt, applies the change to the source model. This modified model is then once again transformed via the graphical model factory and sent to the client for re-rendering.

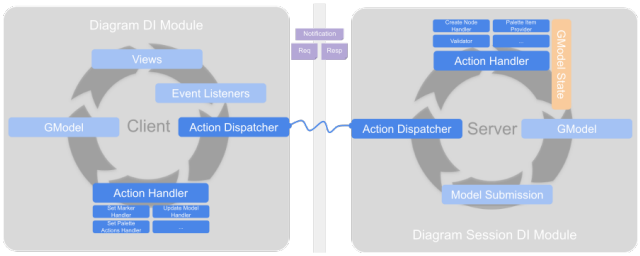


Figure 1: Client/Server action handler cycle in GLSP [10].

In the context of AI, an "agent" refers to a system in which the **Large Language Model (LLM)** serves as the active controller of a (semi-)autonomous system rather than a passive text generator. According to the comprehensive framework proposed by Wang et al. [40], an LLM-powered agent typically comprises four essential modules:

- **Agent Core (Brain:)** The brain serves as the primary reasoning engine. It is prompted with instructions or goals and determines the necessary steps to achieve them. Compared to standard LLM interactions, which require user feedback after each turn, agents operate in a feedback loop until they deem the desired result has been achieved.
- **Planning:** The agent breaks down complex tasks into smaller sub-goals, which enables the model to think step-by-step.

Some agents also incorporate self-reflection, enabling them to critique their past actions and correct errors in real time.
- **Memory:** Short-term memory is managed through the model's context window, which stores the current conversation history. However, long-term memory is often implemented using external vector databases.
- **Tool Use:** By calling external APIs, executing code, or using specialized protocols like the **Model Context Protocol (MCP)**, the agent can reach beyond its closed feedback loop and affect more than just the generation of text or image artifacts.

By integrating these components, LLM agents can become more than simple chat interfaces and perform complex tasks in software engineering, scientific research, and graphical modeling. This transitions AI from a generative tool to an agentic tool, capable of modifying existing artifacts rather than continuously regenerating them, ultimately helping to scale LLM-based intelligent modeling assistants to realistic scenarios [7] rather than regenerating seemingly unrelated parts of the model in subsequent stateless prompt interactions. The agentic use of LLMs provides the additional support needed to improve reliability. This is especially the case in fully deterministic environments, such as the usage of software APIs, where stochastic elements increase the risk of erroneous utilization.

2.2 Model Context Protocol (MCP)

MCP [2] is an open standard, client-server protocol introduced by Anthropic in late 2024 to standardize how LLMs interface with external data and functionality. It is the emerging de facto standard for integrating existing software components with AI agents by providing a universal interface without additional glue code.

MCP's architecture (cf. Fig. 2) is divided into three core entities: the *Host* orchestrates the user session and provides the user interface; it handles *Clients* that maintain a 1:1 stateful connection with an *MCP Server*, which is an independent process exposing functionality to LLM Agents. In our case, the host could either be a desktop agent application like Claude Code or Gemini CLI, or an online LLM integration into a platform like Eclipse Theia, while the server is integrated into the larger GLSP server application structure.

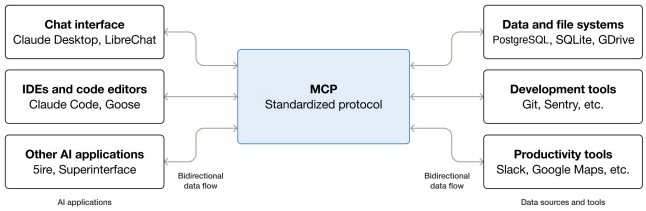


Figure 2: Model Concept Protocol Paradigm [2]

An MCP server provides three key primitives:
- **Resources**: Structured data (e.g., file contents, database schemas, or API documentation) that the server provides to ground the LLM's responses. These act similarly to HTTP GET endpoints.
- **Tools**: Executable functions that the LLM can invoke to perform actions via the API (e.g., creating a file, running a script, or querying an API). These act similarly to HTTP POST endpoints.
- **Prompts**: Pre-defined templates or instructions that the server provides to guide the LLM's behavior for specific tasks.

The protocol uses either stdio for local processes with access to local file systems and tools, or HTTP for remote, network-based services. Communication begins with a negotiation phase that ensures the client is aware of the server's supported features.

This approach provides several advantages. By moving business logic and data to the server, or rather leaving it there, MCP prevents the context window of the LLMs from being overloaded, reducing token costs and increasing accuracy. Additionally, the separation from underlying processes or APIs enables the MCP server to define APIs tailored for LLM use. Also, unlike one-off API calls, MCP supports stateful sessions and sampling, allowing a server to request that the model performs a completion or provide additional information back to the tool.

2.3 Alternative Protocols

MCP is not the only protocol that facilitates and standardizes agentic LLM interactions. This section explores other protocols in this space. Most of them complement MCP's role in the agentic stack rather than competing with it.

2.3.1 Agent2Agent (A2A) Protocol. The Agent2Agent (A2A) Protocol [13] is an open-source communication standard that facilitates direct, secure, and interoperable interactions between autonomous AI agents. It was developed by Google before being donated to the Linux Foundation and incorporating IBM's efforts in the Agent Communication Protocol (ACP) [21]. A2A takes a decentralized approach, enabling agents to discover one another and collaborate on complex tasks without requiring a centralized orchestrator or a shared, proprietary framework.

A2A does not compete with MCP, as both operate at different conceptual levels of the agentic stack. MCP is primarily a "host-to-tool" protocol designed to provide a model with the context and API access within a single execution environment, while A2A is a "peer-to-peer" protocol designed for the collaboration of independent agentic entities. Thus, MCP specifies how an agent uses its tools, while A2A specifies how an agent works with other agents.

2.3.2 Universal Tool Calling Protocol (UTCP). The Universal Tool Calling Protocol (UTCP) [35] is a lightweight, open standard that enables AI agents to discover and interact with tools directly via their native protocols, such as HTTP, gRPC, or Command-Line Interfaces (CLIs). It provides agents with the necessary instructions to communicate with existing endpoints without requiring dedicated wrapper servers. Communication is enabled by JSON-based definitions, known as *Manuals*, that describe the available tools, their input/output schemas, and their native endpoints. Once an agent discovers a tool via such an *UTCP Manual*, the *UTCP Client* handles the execution by calling the tool's native API directly, effectively removing the overhead after the initial discovery phase. The key strengths of this protocol are the elimination of the engineering overhead of maintaining an intermediary server to handle agent requests. This subsequently allows the agent to reuse the tool's

original authentication, billing, and rate-limiting systems, just as any other client.

This contrasts strongly with MCP, whose clear goal is to implement an intermediary layer that the LLM interacts with, which then translates and forwards requests to the actual tools. Although MCP provides a standardized, secure boundary that is often necessary in enterprise environments, it requires additional maintenance, whereas UTCP avoids this by relying on direct access.

3 MCP Systematic Literature Review

Since the inception of MCP, several academic applications, emerging architectural patterns, and implementation challenges have been reported. In the following, we report these findings from a systematic literature review (SLR). The SLR aims to inform the later MCP-GLSP design by responding to the following research questions:

RQ1: What are the existing architectural patterns for implementing MCP servers in software applications?
RQ2: How are the core MCP primitives (Resources, Tools, and Prompts) structured and orchestrated in current implementations?
RQ3: What technical challenges arise when integrating MCP servers with software architectures?

Our search query was intentionally held generic by searching for papers having the term 'model context protocol' in their title or abstract. To filter for relevant papers, we focused on those reporting the development of MCP-enabled systems, primarily MCP servers. This requires that the architectural aspects of the system, its application context, and its implementation details (e.g., configuration and logic of at least one core MCP primitive (Resources, Tools, or Prompts), including custom schemas or mapping strategies) be covered. Additionally, given the emergence of MCP in late 2024, we only considered papers published since 2025. Eventually, we focused on English-language papers.

3.1 Executing the Search

Executing the search on the Scopus, ACM DL, and IEEE Xplore databases yielded a total of 119 papers. After de-duplication, this was reduced to 89 unique papers, 81 of which were actually available. The retrieved papers were filtered using a multi-stage approach. First, an initial review of titles and abstracts, followed by a cursory overview, reduced the pool to 32 candidates; a rigorous full-text evaluation then yielded 12 papers. It is worth noting that many of the papers filtered in the initial screening were not primarily rooted in Computer Science; thus, the focus on documenting the software engineering process was naturally secondary. Additionally, a lack of "model context protocol" or "MCP" in the title is indicative of the lack of importance of MCP in the paper itself. Some papers ostensibly focusing on MCP would actually propose agentic architectures that rely on an MCP server but provide insufficient detail and were thus disregarded. There were also several papers that simply showcased an application of MCP without presenting any interesting changes, emerging patterns, lessons learned, or appropriate metrics, and these were subsequently discarded.

3.2 Relevant Findings

Following the scoping and the SLR's intention to inform the conceptual design of our own MCP server, we report the key findings below, specifically relevant for our architectural considerations. A comprehensive discussion of the various MCP use cases reported in the literature is beyond the scope of this paper. Table 1 summarizes the results of the review with the following columns:

- **Paper ID:** The unique bibliographic reference for the analyzed paper.
- **Context:** A column identifying both the *Application Domain* and the specific *Wrapped System*.
- **Architectural Pattern (RQ1):** Classifies the high-level system topology. While the majority of implementations follow a *Standard Wrapper* approach, unique architectures are explicitly highlighted as such.
- **Primitives (RQ2):** Indicates which core MCP components are used in the implementation: Tools (T), Resources (R), and Prompts (P).
- **Primitive Strategy (RQ2):** Describes the design philosophy behind the primitives.
- **Client Orchestration:** Details any special logic or usage of the MCP Client, including prompting strategies, agentic control loops, or the use of fine-tuned models.
- **Challenge Encountered (RQ3):** Summarizes the primary technical challenge addressed by the paper.

Answer to RQ1 (Architectural Patterns): Most works follow a standard MCP architecture in which a single server wraps an existing software system and exposes its functionality via tools (e.g., [3, 5, 8]). This pattern proves effective for integrating LLMs into established workflows by enforcing validation and structured interaction. However, several variations emerge. Huang et al. [20] demonstrate a *separation-of-concerns* approach that uses multiple MCP servers (e.g., for documentation and templates), thereby improving performance over monolithic or RAG-based solutions. Similarly, Guo et al. [14] distribute functionality across multiple servers representing heterogeneous devices, enabling intent-driven interaction in IoRT environments, albeit at the cost of increased semantic ambiguity. More dynamic approaches, such as SensorMCP [15], extend this idea further by enabling *runtime tool generation*, highlighting the flexibility of MCP beyond static wrappers. These findings suggest that while the standard architecture is dominant, more modular and adaptive designs can improve expressiveness and task performance when carefully controlled.

Answer to RQ2 (Primitives): Across the analyzed works, MCP implementations predominantly rely on *Tools*, with limited use of *Resources* and minimal discussion of *Prompts*. This is partly due to incomplete client-side support for resources, leading to tool-centric designs that are more broadly applicable. Several authors advocate for *small, well-defined toolsets* to reduce cognitive load and ambiguity (e.g., [20, 39]), showing that compact APIs improve reliability. However, other approaches successfully employ larger, more structured collections of tools (e.g., [41]), particularly when combined with state-aware orchestration. In some cases, tools encapsulate complex workflows or even external reasoning processes, such as LLM-mediated retrieval [23] or structured parsing of legacy outputs [32]. Overall, the results indicate that the effectiveness of

Table 1: MCP Systematic Literature Review Extraction Table

Paper ID	Context (Domain / Wrapped System)	Architect. Pattern	Prim.	Primitive Strategy	Client Orchestration	Challenge Encountered
Huang et al. [20]	Meta-Optics *TorchRDIT*	**Dual-Server Setup**	T	Small API of 5 tools; Separates code templates from documentation into separate MCP servers.		Niche solver documentation is insufficient for LLMs; Context window limits prevent full manual dumping.
Avila et al. [3]	Structural Eng. *OpenSeesPy*	Standard Wrapper	T	Stateless Modules; Tools separated by function (Modeling, Sim, Valid).		Safety-critical domain requires strict numeric validation.
Szeider [39]	Comb. Opt. *Z3, PySAT, MiniZinc*	Standard Wrapper	T	Incremental Editing; Tools designed for "item-based" model editing reducing the number of tools.	**ReAct Loop:** Iterative solving with an agentic loop + separate "Reviewer Agent".	LLMs fail at complex logical reasoning.
Wu & Barnett [41]	3D Authoring *Unity Editor*	Standard Wrapper	T, R	Resources provide a read-only scene graph; a medium number of tools categorized by hierarchy.	**State-Aware Chat:** Client injects current scene state with every request to ground the LLM.	LLMs lack spatial/state awareness of the 3D scene; LLMs lack complex spatial reasoning.
Higuchi et al. [18]	Web A11y *Biome*	Standard Wrapper	T	3 tools; 1 tool retrieves specific accessibility rules/docs alongside the analysis tool.	**Lint & Fix-Loop:** Agent analyzes error → fetches rule docs → applies fix → re-validates.	LLMs generate superficial fixes or introduce new errors without access to specific rule documentation.
Matsuda et al. [23]	Cybersecurity *Elasticsearch, Sigma*	**MCP-for-LLM**	T	Tools use another LLM to search the internet for rules; Tools search domain-based rules; Access database with informed queries.		High false positive rate in log analysis; Difficulty distinguishing malicious acts from admin tasks.
Guo et al. [14]	IoRT (Robotics) *Edge Sensors*	**Multi-Server Setup**	T, R, P	Each Device is a separate MCP server; Sensors → Resources; Actuators → Tools; Workflows → Prompts.		Semantic ambiguity across heterogeneous devices; Real-time latency requirements.
Guo et al. [15]	Wildlife Monitoring *Smart Cameras*	**Dynamic Tool Factory**	T	Server creates *new* tools at runtime; Generic tool invocation; 3 tools in total.		Manual tool creation is unscalable; General-purpose models lack context for specific sensing tasks.
Bandara et al. [5]	Blockchain *Smart Contracts*	Standard Wrapper	T	Direct mapping of Smart Contract functions to MCP Tools.	**Fine-Tuned:** Llama-4 fine-tuned to extract technical parameters for MCP signatures.	Protocol mismatch between probabilistic LLMs and deterministic/immutable Blockchain ledgers.
Bandara et al. [4]	5G Networks *Open5GS*	Standard Wrapper	T	Stateful tools; Tools encapsulating complex slice lifecycle logic.	**Fine-Tuned:** Llama-4 fine-tuned to extract technical parameters for MCP signatures.	Complex orchestration across heterogeneous domains; High cost of configuration errors.
Chen et al. [8]	Maritime Ops *AISStream, Neo4j*	Standard Wrapper	T	Modular toolkit; Distinct tools for Live Stream vs. Historical Graph Query.		Ambiguous data streams lead to inconclusive results; Complex reasoning fails with simple tool use.
Qiu et al. [32]	Legacy Opt. *Gurobi*	**Local Server**	T	Output suppression of used software; Wrapper parses verbose legacy stdout into structured JSON.	**Robust Error Handling:** Multi-layer rollback mechanism to handle solver failures.	Legacy software relies on unstructured stdout/stdin, polluting the JSON-RPC communication channel.

primitives depends less on their number and more on their *clarity, separation of concerns, and alignment with the task structure.*

Answer to RQ3 (Challenges): A recurring challenge is that MCP integration does not inherently improve LLMs' reasoning capabilities. While MCP enables access to deterministic tools that compensate for known LLM weaknesses (e.g., symbolic reasoning [39] or numerical validation [3]), complex reasoning tasks still require careful orchestration or external support. Increasing the number of tools or servers introduces additional risks, particularly *semantic ambiguity* between similar tool descriptions, which can degrade performance [14]. Furthermore, MCP assumes clean, structured communication channels; legacy systems relying on unstructured stdout require additional preprocessing to avoid protocol violations [32]. Finally, context limitations remain a bottleneck, especially when large documentation or model representations are required [20].

Beyond these challenges, several cross-cutting insights emerge. First, MCP-based integration often enables general-purpose LLMs to outperform specialized models when combined with appropriate tool support, reducing the need for costly fine-tuning. Second, a key motivation across all works is to lower the barrier to entry for complex systems by enabling intent-driven interaction, replacing the need for deep domain expertise and programming skills with structured natural language interfaces.

Taken together, these findings highlight that the design of MCP-based systems requires careful balancing of architectural structure, primitive design, and orchestration strategies. In particular, they point to the importance of providing structured, semantically grounded access to complex system models—an aspect that is not yet systematically addressed in existing work. The following sections build on these insights by designing, implementing, and evaluating an MCP server tailored to graphical modeling environments based on the Graphical Language Server Platform (GLSP).

4 MCP-GLSP

Whereas the current research focuses on the generative use of LLMs, AI Agents, as outlined by Wang et al. [40], seem severely underutilized despite their significant potential to alleviate current problems. Mazur et al. [24] show that using an agent with access to simple file-system tools, rather than the entire serialized model, significantly reduces the context size, though at the cost of some performance degradation. However, using fit-for-purpose tools may improve the performance while keeping the context small.

Additionally, rather than the previous *Generative* paradigm, which regenerated a model with each round of modifications, this *Agentic* paradigm does not directly generate a model but instead interacts with it via a given API. While this restricts the agent's possibilities to those allowed by the API, the lack of direct modification serves as a safeguard against unintended changes. This means that a modification does not entail rewriting the entire model, but only calling the appropriate tool for effecting the change. The much smaller context also promises to improve scalability.

This shows a distinct lack of research on using LLM Agents to support modeling efforts, and even less on using standardized protocols, such as the Model Context Protocol (MCP), to facilitate *Agentic Model Interaction*. This research aims to move the paradigm for using LLMs in MDE forward from *Writing* to *Interacting*. The following transparently reports our reflections on iterative development and testing cycles, leading to the final architecture and implementation of MCP-GLSP.

4.1 Architecture

In the following, we outline the target architecture for implementing an MCP server into GLSP's existing client-server architecture (see Fig. 3). It is composed of three central components:

- **Browser Client:** This represents any kind of browser or browser-like application that can run GLSP's frontend code. Whether this is an actual browser, VS Code, Eclipse Theia, or something different does not matter. Each opened diagram constitutes a separate **GLSP Client** and corresponds to a **Session** on the **GLSP Server**.
- **LLM Client / MCP Host:** This component matches the role of *Host* in the MCP architecture. This could be an LLM CLI tool, like Gemini CLI or Claude Code. Each chat or session on the host is a separate *Client* under the MCP architecture, which establishes a 1:1 connection to the *MCP Server* within a **Session**.
- **NodeJS Server:** The server is the heart of the architecture, handling both **GLSP Server** and **MCP Server**. The former is part of the core GLSP application, while the latter is an addition. Therefore, the **GLSP Server** starts the **MCP Server** within the same dependency context, which allows access to the **ClientSessionManager**. This subsequently grants access to the **GLSP Server**'s **Sessions**. All components run in the same process.

The entire workflow can be summarized as follows:

(1) The **NodeJS Server** is started.
(2) Some client application (e.g., Eclipse Theia) connects to the **NodeJS Server**, which starts the **GLSP Server** and the **MCP Server**.

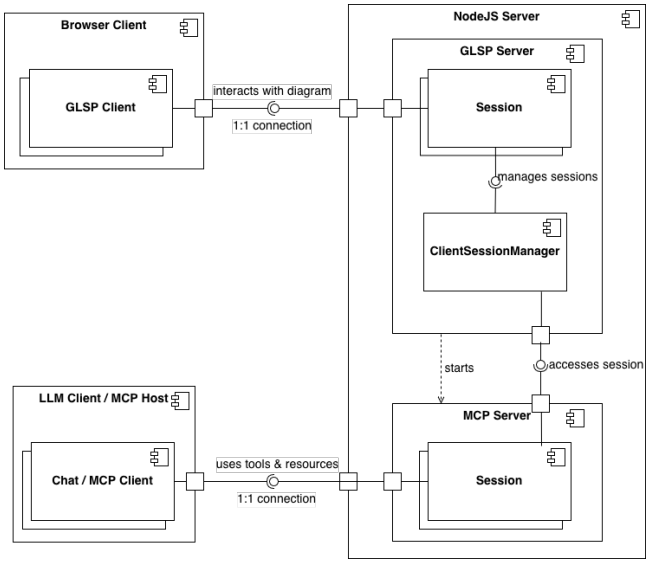


Figure 3: GLSP-MCP Architecture

(3) A user starts a **GLSP Client**, i.e., opens a diagram/model.
(4) This client connects to the **GLSP Server**, causing the creation of a **Session** with access to the modeling context. Furthermore, the **Session** is registered with the **ClientSessionManager**.
(5) A user wants to use an **LLM Client**, such as Gemini CLI, to act as a modeling assistant, starting a **Chat** (or rather, **MCP Client**) that connects to the **MCP Server**. The server generates a **Session** (distinct from those in GLSP) to track the interaction.
(6) Depending on the conversation in the **Chat**, certain tools are invoked. This means that the **MCP Session** uses the **ClientSessionManager** to access the corresponding **GLSP Session** and thus obtain the modeling context. The tool can now effect changes in the model via actions, injecting itself into the GLSP event cycle.
(7) The result of most actions is usually a response action, which is sent back to the **GLSP Client** to inform it of the changes and enable graphical display.

4.2 Implementation Decisions

This section presents the implementation of the MCP server for GLSP as a reusable software artifact. The implementation focuses on enabling reliable and efficient interaction between LLMs and graphical modeling environments while maintaining generalizability across different GLSP-based languages. Several key implementation decisions emerged during iterative development and are valuable to other MCP server developers.

Artifact Design and Architecture The core artifact is a modular MCP server that exposes GLSP functionality via structured endpoints. Each endpoint encapsulates both its interface (schema) and implementation logic, while a generic orchestrator dynamically registers all available endpoints using dependency injection. This design ensures extensibility and allows domain-specific adaptations without modifying the core infrastructure.

To maximize compatibility across MCP clients, all functionality is exposed via *Tools*, with optional embedding of resource-like data. This avoids reliance on partially supported MCP primitives and ensures the artifact's portability across different LLM ecosystems.

Efficient Representation and Context Optimization A key implementation concern was reducing the size and complexity of the model context exchanged with the LLM. Replacing verbose XML or JSON serializations with a lightweight markdown/text representation reduced token usage by 30–60% while preserving semantic clarity. Similarly, replacing UUIDs with short, semantically enriched identifiers improved both efficiency and reference accuracy. These optimizations are essential because nearly all MCP interactions depend on the model context. The results (cf. the evaluation section for details) indicate that compact, semantically structured representations are preferable over formally complete but verbose encodings.

Shifting Computation to Deterministic Components The implementation systematically shifts computational tasks from the LLM to deterministic server-side components. For example, spatial constraints such as element bounds are precomputed and explicitly provided, rather than requiring the LLM to derive them. This reduces both token usage and inference time by up to 50%. Similarly, computationally intensive tasks such as layouting are delegated to dedicated algorithms exposed via MCP tools. Compared to LLM-based reasoning, this reduces execution time and cost by up to 85%, highlighting the importance of combining LLM reasoning with deterministic processing.

Tool Design and Invocation Strategy The design of MCP tools follows two principles: minimizing ambiguity and reducing invocation overhead. In addition to exposing fine-grained GLSP operations, higher-level composite tools are provided that internally orchestrate multiple actions. Additional tools are designed to operate on multiple elements simultaneously (multi-target tools), thereby reducing the number of MCP calls required. Our empirical observations (cf. Section 5) show that reducing tool invocations can decrease both cost and execution time by approximately 50%, indicating that MCP call overhead is a significant factor in system performance.

Abstraction Levels: GLSP Actions vs. MCP Tools A central design challenge was reconciling the different abstraction levels of GLSP and MCP. GLSP is built around fine-grained, interaction-oriented actions (e.g., creating an element at a position, updating a label), reflecting fine-grained user-driven graphical editing. In contrast, MCP tools are invoked through natural language and are expected to perform higher-level, intention-driven operations. Directly exposing GLSP actions as MCP tools proved insufficient, as LLMs tend to operate at a higher level of abstraction and struggle to correctly sequence multiple low-level actions. This mismatch led to inefficient interaction patterns and unnecessary tool calls. To address this, the implementation introduces *composite MCP tools* that encapsulate multiple GLSP actions into a single semantic operation (e.g., creating and naming an element in one step). This reduces interaction complexity and aligns better with LLM reasoning capabilities.

Several obstacles emerged in this context. First, composite tools execute multiple commands internally, each of which is individually tracked by the GLSP command stack. This creates inconsistencies for undo/redo operations, as the LLM perceives a single action while the system records multiple ones. This was mitigated by exposing metadata about executed commands and enabling parameterized undo operations. Second, LLMs often ignored the immediate outputs of tools and redundantly queried additional endpoints. This was resolved by introducing dedicated, fine-grained retrieval tools (e.g., element detail queries), reducing the need for full model retrieval and improving interaction efficiency.

Overall, this abstraction layer is critical for bridging human-like intent and system-level execution, and represents a key contribution of the artifact.

Integration with GLSP and Communication Model Integrating MCP into GLSP required bridging asynchronous event-based communication with synchronous MCP request–response interactions. This was achieved by combining event-driven mechanisms with promise-based synchronization, enabling complex workflows (e.g., model export) to be handled within a single MCP request. While effective, this approach introduces challenges for concurrency and multi-session scenarios, which require further investigation in future work.

4.3 MCP Server Structure

Next, all resources and tools available on the MCP-GLSP server are documented, including their parameters, expected inputs, and return values.

4.3.1 Resources. Resources in the MCP protocol are URI-addressed, read-only data sources that an agent may read at any time. The MCP-GLSP server defines four resources, each identified by a `glsp://` URI scheme. Since not all MCP clients support the resource protocol, the server can be configured to expose these endpoints as regular tools instead.

`sessions-list` (`glsp://sessions`) This resource provides a snapshot of all active GLSP client sessions at the time the request is made. Each session entry contains the session identifier, the diagram type being edited, the URI of the source model file, and a flag indicating whether the session is in read-only mode. The result is formatted as a Markdown table.

This resource is typically the first point of interaction for an agent, as the session identifier returned here is required by all other tools and resources.

`element-types` (`glsp://types/{diagramType}/elements`) This parameterized resource returns a catalog of all element types that can be created in diagrams of the specified type. By default, the list is dynamically derived from the GLSP server's `OperationHandlerRegistry` at runtime and is therefore always consistent with what the server actually supports. However, any implementation of GLSP may likely want to override and rebind this handler to provide information more relevant to an implemented diagram type.

Results are separated into node types and edge types and presented as Markdown tables, each row containing the element type identifier and a human-readable label. At least one active session of the requested diagram type must exist for this resource to be resolvable.

`diagram-model` (`glsp://diagrams/{sessionId}/model`)
This resource serializes the complete graph model of the specified session into a structured Markdown representation. The serialization covers the entire model tree, including all nodes, edges, their nesting relationships, positional bounds, and additional properties. Identifiers are substituted with their short aliases when ID aliasing is enabled. The result serves as the primary source of truth for an agent that needs to understand the diagram's current state before making modifications.
As this is necessarily a generic serialization, GLSP implementations may want to override and rebind this handler (or rather, the `McpModelSerializer`) to optimize the model's representation.

`diagram-png` (`glsp://diagrams/{sessionId}/png`) This resource triggers an asynchronous PNG export of the diagram by dispatching a dedicated action to the connected frontend client and waiting for a response. Because visual rendering logic resides in the frontend, this resource depends on an active client connection and imposes a five-second timeout. The returned PNG data is base64-encoded and particularly useful for visually verifying layout decisions made by an agent.

4.3.2 Tools. Tools are callable operations with a defined input schema that an agent may invoke to either query the diagram for information or apply changes to it. Unlike resources, tools are stateful in the sense that each modifying tool changes the model. Table 2 gives an overview of all available MCP-GLSP tools grouped by their functional category.

All tools except `sessions-list` and `element-types` require the parameter `sessionId`. It must match a session identifier from the `sessions-list` resource. When ID aliasing is active, all element identifier parameters must be alias strings returned by prior tool calls; the server transparently resolves them to real identifiers.

Table 2: All tools provided by the MCP-GLSP Server

Tool Name	Category	Brief Description
`sessions-list`	Discovery	Lists all active GLSP client sessions.
`element-types`	Discovery	Lists creatable element types for a diagram type.
`diagram-model`	Inspection	Retrieves the full serialized model of a session.
`diagram-elements`	Inspection	Retrieves serialized details of specific elements.
`diagram-png`	Inspection	Returns a PNG rendering of the current diagram state.
`get-selection`	Inspection	Returns the IDs of currently selected UI elements.
`create-nodes`	Modification	Creates one or more node elements in the diagram.
`create-edges`	Modification	Creates one or more edge elements between nodes.
`delete-elements`	Modification	Deletes one or more elements, including dependents.
`modify-nodes`	Modification	Changes position, size, or label of existing nodes.
`modify-edges`	Modification	Changes the endpoints or routing of existing edges.
`save-model`	Persistence	Persists unsaved changes to the source model file.
`undo`	History	Reverts the last n executed commands.
`redo`	History	Re-applies the last n undone commands.
`validate-diagram`	Validation	Runs validation and returns diagnostic markers.
`change-view`	View	Changes the viewport of the connected UI client.
`request-layout`	View	Triggers automatic layout (optional, see below).

5 Evaluation

The evaluation subsequently presented aims to shed light on the MCP-GLSP solution from two perspectives: the one of GLSP tool developers interested in integrating and customizing the MCP server to transform their conventional GLSP-based modeling editor into an intelligent modeling assistant; and the perspective of MDE and AI researchers interested in understanding the effect of using an MCP server on tool-LLM-interaction and downstream tasks. Admittedly, this latter evaluation is not free from threats to validity, such as the use of a single LLM and a single modeling language, as well as limited statistical rigor (only 5 runs). However, we believe this broad initial evaluation offers many interesting insights and demonstrates efficacy.

5.1 The Tool Developer Perspective

To assess the practical applicability of the proposed MCP–GLSP solution, we integrated the generic MCP server into the existing GLSP-based UML modeling editor BIGUML [25]. The goal was to evaluate both the required effort and the necessary adaptations for a real-world language implementation.

Mandatory Integration. The core integration is minimal and requires only enabling the MCP modules during server initialization. By adding the MCP init and server modules to the GLSP launcher, all default MCP endpoints become immediately available. In the case of BIGUML, this required fewer than 10 lines of code and yielded a largely functional MCP interface out of the box.

Moreover, only minor adjustments were necessary to ensure correctness: (i) disabling unsupported handlers (e.g., edge modification if not implemented in the editor), (ii) adapting label resolution logic to the language-specific model structure, and (iii) configuring the client environment (e.g., content security policies for image handling). These adjustments are lightweight and typically remain below 100 lines of code.

Optional Adaptations. While the default integration is sufficient to enable an MCP server for arbitrary GLSP-based modeling editors, we learned that the following optional extensions and configurations can significantly improve usability and efficiency:

- **Custom model serialization:** The generic serializer produces verbose and weakly structured outputs. Replacing it with a modeling language-specific serializer reduces context size and improves semantic clarity.
- **Enhanced element type descriptions:** Default type listings rely on internal identifiers, which often do not carry meaningful semantics. Curating descriptions of modeling elements improves LLM guidance and reduces ambiguity.
- **Deterministic tool support:** Integrating layouting or other available algorithmic tools allows the LLM to delegate computational tasks, improving performance and reliability.
- **Custom MCP handlers:** Additional tools or resources can be easily added by registering new handlers, enabling domain-specific extensions without modifying the core infrastructure.

Lessons Learned. The integration demonstrates that MCP–GLSP is highly reusable and requires minimal effort for basic functionality. At the same time, optimal performance depends on language-specific adaptations, particularly regarding model representation

and semantic enrichment. A key observation is that the generic solution provides a strong baseline, while targeted customization—especially of serialization and type descriptions—yields substantial improvements. Overall, the case study demonstrates that the MCP server can be seamlessly integrated into GLSP-based editors and provides a practical foundation for AI-assisted modeling.

5.2 The MDE and AI Researcher Perspective

Next, we analyze how key design and implementation decisions influence the performance of downstream tasks. In all reported experiments, we used Claude Sonnet 4.6 via Claude Code (v2.1.86) as an AI agent. The evaluation is conducted using a GLSP Workflow model[1] as a representative modeling language.

5.2.1 Research Objective 1: Scalability. We evaluate four configurations combining two core factors: (i) **representation format** (Markdown vs. JSON), and (ii) **identifier scheme** (semantic aliasing vs. UUIDs). Markdown provides a compact representation of structured, tabular data in the form of Markdown tables, which needs no redundant keywords compared to a list of the same-structure JSON objects (cf. Listing 1). ID aliasing refers to replacing each occurrence of an ID in MCP communication with a simple integer, with the mapping between them stored per session (cf. Listing 2).

Listing 1: JSON with UUIDs

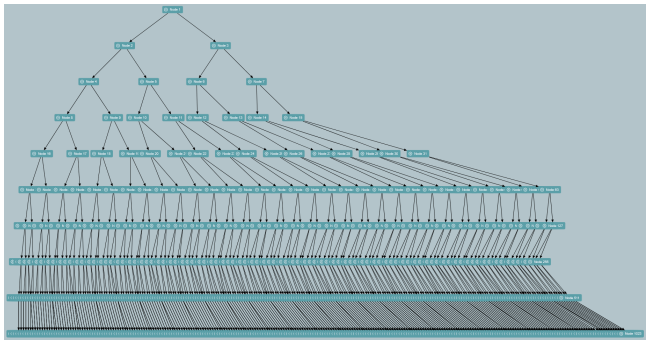
```
{
  ...
  "task:automated": [
    {
      "id": "8bec1e4d-6154-4845-9edb-f55512b8d892",
      "position": {
        "x": 479,
        "y": 210
      },
      "size": {
        "width": 163,
        "height": 30
      },
      ...
    },
    {
      "id": "cff5f5e0-5c3a-400c-aa17-ddeef46a0b32",
      ...
    },
    ...
  ],
  ...
}
```

Listing 2: Markdown with ID aliasing

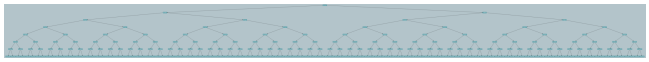
```
# task:automated
| id | position | size | ... |
| --- | --- | --- | ... |
| 2 | x:479,y:210 | width:163,height:30 | ... |
| 3 | x:139,y:210 | width:122,height:30 | ... |
...
```

Each configuration is evaluated across five runs to assess consistency and investigate variance. Tasks simulate common modeling activities and are designed to stress scalability and reasoning capabilities:

- **Model creation:** incremental construction of a binary tree of model elements (see Fig. 4a) (scalability stress test),
- **Model modification:** global alignment without algorithmic support (reasoning-intensive task) (see Fig. 4b),
- **Model analysis:** extraction of structural model properties (information retrieval task).



(a) Maximum Depth Binary Tree Graph for Markdown and Aliasing



(b) Aligned Binary Tree Graph for Markdown and Aliasing

Figure 4: Scalable evaluation scenario

Key Findings. Table 3 shows the evaluation results structured as follows: **Configuration** is the chosen feature set, and **Levels** shows both how many levels can be created under the constraints and how many are viable, i.e., produce small enough serializations to fit into the context. **Elements** is the number of maximally possible elements that can be handled using a given feature set. **Cost**, **Time**, and **Tokens** display the average values across all three **Tasks**. The tokens are further divided into output, cache-read, and cache-write. Based on the results presented in Table 3, we derive the following key findings.

(1) Context size is the primary bottleneck. Across all configurations, the maximum manageable model size is constrained by the size of serialized model representations. Markdown-based representations consistently reduce context size (30–60%), enabling significantly larger models compared to JSON.

(2) Identifier design critically affects scalability. Replacing UUIDs with short, semantically meaningful aliases yields substantial improvements. This reduces token usage per reference by up to 95%, effectively doubling the number of model elements that can be handled.

(3) Combined optimizations are multiplicative. The combination of Markdown and aliasing nearly doubles the feasible model size compared to baseline configurations, enabling models with approximately 600 elements, whereas unoptimized configurations are limited to 250–350 elements.

(4) Task type influences sensitivity to optimizations. Creation and modification tasks are highly sensitive to representation size, while analysis tasks (reading) are comparatively stable across configurations. This suggests that generation tasks are the primary limiting factor for scalable AI-assisted modeling.

[1] https://eclipse.dev/glsp/examples/#workflowoverview

Table 3: Feature Set Comparison based on Creation, Modification, and Analysis Scenarios

Config.	Levels		Elements	Task	Cost	Time	Tokens		
	Created	Viable					Output	Read	Write
Markdown and Aliasing	10	8	600	Create	1.70$	620s	72k	966k	93k
				Modify	0.54$	202s	20k	254k	43k
				Analyze	0.13$	23s	1k	84k	22k
Markdown without Aliasing	9	7	350	Create	2.08$	671s	75k	1.28m	154k
				Modify	0.67$	392s	24k	135k	71k
				Analyze	0.13$	24s	1k	76k	23k
JSON and Aliasing	10	7	330	Create	1.63$	600s	67k	945k	91k
				Modify	0.36$	150s	14k	171k	27k
				Analyze	0.09$	25s	1k	82k	14k
JSON without Aliasing	8	6	250	Create	1.55$	982s	62k	1m	80k
				Modify	0.32$	180s	12k	152k	25k
				Analyze	0.10$	27s	2k	76k	13k

(5) Hard limits emerge at the system level. Independent of LLM performance, MCP tool responses exhibit size limits (≈ 50k characters), imposing an upper bound on model complexity. This highlights the need for future work on incremental or batched context retrieval.

Implications for AI4MDE. The results demonstrate that representation design is not merely an implementation detail but a determining factor for the feasibility of AI-assisted modeling. Efficient, semantically structured representations are essential to enable large-scale model manipulation and should be considered a first-class design concern in AI4MDE systems. MCP–GLSP supports tool developers by providing initial functions to support this and extension points for language-specific adaptations.

5.2.2 Research Objective 2: Interaction Paradigms. In this part of the evaluation, we focus on the extent to which LLM-assisted modeling with structured MCP integration compares with existing serialization-based interaction methods in terms of retrieval accuracy, model quality, and task type.

Experimental Setup. We comparatively evaluate three possible modeling interactions:

MCP-based interaction: structured tool-based interaction with a graphical modeling environment,

JSON-serialization-based interaction: direct manipulation of JSON model encodings,

XML-serialization-based interaction: direct manipulation of XML model encodings.

In this evaluation, we use four tasks to simulate common modeling activities. Since Workflow modeling is a subset of BPMN, publicly available modeling questions for BPMN [38] are used and adjusted, with **3 tasks** selected based on increasing size and complexity. In addition, a modification task (**Task 3.5**) is introduced that explicitly assesses the ability to manipulate an existing model rather than create a new one. Furthermore, **Task 4** tests the ability to handle particularly large models under precise, step-by-step instructions[2]. Both the semantic and the graphical information of the model is communicated at once. While this may marginally affect the quality of purely semantic requests, the main purpose of this integration remains to support graphical modeling.

The AI Agent is asked to respond to the following questions:

- List the nodes that exist in the model by type?
- What are the start and end tasks?
- How many diverging decisions or parallel paths exist?

The performance of the AI Agent is assessed along two dimensions: **(1) Information Retrieval (Reading)** is assessed using Precision and Recall. The **(2) Model Construction and Modification (Writing)** performance is assessed via syntax errors (i.e., violations of modeling language constraints); Semantic Errors (i.e., deviations from intended model structure); and Pragmatic Errors (i.e., reduced clarity or unnecessary complexity). Additionally, costs and API interaction times are measured to capture the efficiency with which tasks are handled. To ensure comparability across task sizes, error counts are normalized relative to the expected model size. Error metrics are calculated by comparing a generated output model with a human-generated ground truth solution. However, as not all models must look exactly the same to be semantically equivalent, some leniency is given.

5.2.3 Results. Table 4 shows that the MCP-based interaction paradigm maintained 100% precision and recall and scaled well with increasing model size, both in terms of cost and time. The serialization-based approaches, while still delivering solid results, instead scale much worse with model size. It can be observed that, once a certain size is reached, the LLM employs Python scripts and behaves like a system that extracts relevant information, resulting in good scores. This intermediary step is unnecessary for MCP, which only needs to handle the protocol's communication overhead.

Table 4: Results of the Information Retrieval Experiments

Task	Format	Precision	Recall	Cost ($)	Time (s)
Task 1	JSON	0.988	1.000	0.055 ± 0.014	27 ± 2
	XML	1.000	1.000	0.048 ± 0.009	28 ± 2
	MCP	1.000	1.000	0.064 ± 0.005	33 ± 3
Task 2	JSON	1.000	1.000	0.119 ± 0.025	54 ± 11
	XML	0.996	1.000	0.121 ± 0.020	67 ± 8
	MCP	1.000	1.000	0.069 ± 0.003	38 ± 3
Task 3	JSON	0.970	0.996	0.127 ± 0.037	75 ± 39
	XML	0.887	0.993	0.201 ± 0.063	107 ± 52
	MCP	1.000	1.000	0.093 ± 0.010	44 ± 6
Task 4	JSON	0.982	1.000	0.221 ± 0.071	71 ± 8
	XML	0.943	1.000	0.183 ± 0.033	82 ± 13
	MCP	1.000	1.000	0.159 ± 0.020	64 ± 7

However, when evaluating the error metrics in Table 5, large differences emerge. For serialization-based approaches, both costs and times increase significantly with increasing model size, up to the point of non-functionality, when the output becomes too large for the model's context window (under default settings), and generation fails. Otherwise, serialization-based approaches perform surprisingly well for model creation (Tasks 1-3), with few errors. This suggests that, given a reference model serialization and a task description, they are indeed viable for scaffolding new models. However, as Task 3.5 shows, they perform much worse when modifying an existing model, sometimes even completely deleting the original model. In comparison, the MCP interaction paradigm seems to handle models of varying sizes and complexity equally well, scaling

[2]All task descriptions and the involved models can be found in the online supplementary material of this paper: https://github.com/Sakrafux/mcp4glsp-evaluation-tasks

Table 5: Results of the Model Construction and Modification Experiments

Task	Format	Errors			Norm. Errors			Cost ($)	Time (s)
		Syn	Sem	Prag	Syn	Sem	Prag		
Task 1	JSON	0.300	0.300	0.200	0.023	0.023	0.015	0.310 ± 0.082	137 ± 34
	XML	0.000	0.500	0.700	0.000	0.038	0.054	0.362 ± 0.076	155 ± 20
	MCP	0.000	1.000	0.000	0.000	0.077	0.000	0.193 ± 0.011	94 ± 6
Task 2	JSON	0.000	1.700	3.300	0.000	0.028	0.054	0.662 ± 0.044	340 ± 25
	XML	2.800	1.500	3.000	0.046	0.025	0.049	0.766 ± 0.126	369 ± 67
	MCP	0.000	1.100	1.100	0.000	0.018	0.018	0.308 ± 0.056	161 ± 24
Task 3	JSON	0.000	6.000	0.100	0.000	0.069	0.001	1.194 ± 0.527	740 ± 388
	XML	0.000	2.400	2.200	0.000	0.028	0.025	1.351 ± 0.543	838 ± 378
	MCP	0.000	3.400	0.000	0.000	0.039	0.000	0.422 ± 0.076	231 ± 53
Task 3.5	JSON	4.900	9.400	3.300	0.056	0.108	0.038	0.235 ± 0.064	103 ± 29
	XML	12.400	9.000	0.000	0.143	0.103	0.000	0.673 ± 0.229	249 ± 77
	MCP	0.000	0.000	2.500	0.000	0.000	0.029	0.202 ± 0.019	102 ± 16
Task 4	JSON	––	––	––	––	––	––	2.354 ± 0.329	1455 ± 242
	XML	––	––	––	––	––	––	2.610 ± 0.458	1479 ± 248
	MCP	0.000	0.000	1.800	0.000	0.000	0.005	0.732 ± 0.084	294 ± 35

well in both cost and time. Especially if given precise specifications (Task 4), few issues occur.

It could be argued that the little context given to the LLM about the structure of the Workflow language puts it at a disadvantage. However, our experiments show that the LLMs had few issues with the modeling language, cf. the syntactic errors reported in Table 5. Instead, our experiments show that the LLM struggled to infer the assignment's intent and generate a semantically and pragmatically correct answer (e.g., failing to produce the correct task flow; cf. Table 5). This suggests that the LLM did not miss language 'understanding,' but further experiments are needed to substantiate this.

5.2.4 UML and auxiliary results. While the achieved results are somewhat indicative of the usefulness of MCP for graphical modeling, only using a single, custom modeling language is somewhat limited in meaningfulness. In [16] we explore UML as well using BIGUML as the corresponding GLSP-based implementation. Here, the MCP interaction paradigm also proved capable of solving tasks to a satisfying degree and retrieving information largely correctly. However, the alternative interaction paradigm of using the LLM to generate and modify PlantUML code proved superior in time/-cost efficiency and, to a lesser degree, in output quality. Further research is necessary to better understand the factors influencing the benefits (and limitations) of using MCP-based interaction in graphical modeling in general.

We also investigated the usage of other LLM models, comparing Sonnet 4.6 (the basis of our evaluation) with Opus 4.6, Gemini Flash 3, and Gemini Pro 3.1. All LLMs have shown basic proficiency and proved capable of solving the task, though there were some qualitative differences, with Opus 4.6 performing best and Gemini Flash 3 worst. However, some use cases may not require the highest degrees of quality in favor of faster and cheaper processing. This shows that other, less capable models are viable for use as well.

5.3 Implications for AI4MDE

The evaluation highlights that MCP-based interactions fundamentally change how LLMs interact with models. By delegating execution to deterministic tools, MCP reduces error-prone text generation and improves alignment with modeling semantics. Our results also show that the MCP–GLSP integration particularly outperforms conventional LLM-based modeling when existing models require modifications. However, its effectiveness depends on efficient context management and well-designed tool abstractions, as shown in the feature-level evaluation. Overall, the integration demonstrates that MCP–GLSP can be very effective in improving not only the scalability but also the quality of AI-assisted modeling.

6 Conclusion

This paper introduced an MCP server for the Graphical Language Server Platform that enables LLM agents to interact with graphical models through structured, tool-mediated operations. Moving beyond generative approaches, the proposed solution establishes an interaction paradigm in which LLMs act as modeling assistants rather than model generators. From an artifact perspective, the MCP–GLSP server demonstrates high reusability and low integration effort, as evidenced by its use in two existing GLSP-based editors. The modular architecture and extensibility mechanisms allow developers to adapt the solution to their modeling languages with minimal effort. MCP–GLSP was reviewed by the GLSP developers, and merged into the GLSP core repository, making it publicly available to tool developers[3].

Empirically, the evaluation shows that MCP-based interaction improves downstream modeling tasks. Compared to serialization-based approaches, MCP achieves higher robustness, particularly for model modification tasks, while maintaining excellent retrieval performance and better scalability. These improvements stem from (i) structured access to model context, (ii) delegation of computation to deterministic tools, and (iii) alignment between LLM reasoning and tool abstraction levels.

At the same time, the study highlights important design considerations for AI4MDE systems, including the critical role of representation efficiency, abstraction design, and tool orchestration. MCP does not replace LLM reasoning limitations but provides a principled mechanism to compensate for them through controlled interaction. Future work includes addressing context size limitations through incremental model access, improving multi-session robustness, and extending the approach to additional modeling paradigms and domains. Overall, the presented work demonstrates that protocol-based integration is a key step toward scalable, reliable, and practically usable AI-assisted modeling environments.

Acknowledgments

This study was partially funded by the Austrian Research Promotion Agency (FFG)-funded project Smart GLSP – Facilitating Large Language Models for Smart GLSP-based Modeling (Grant number: FO999925707).

References

[1] Syed Juned Ali, Aleksandar Gavric, Henderik A. Proper, and Dominik Bork. 2023. Encoding Conceptual Models for Machine Learning: A Systematic Review. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023 Companion, Västerås, Sweden, October 1-6, 2023.* IEEE, 562–570. doi:10.1109/MODELS-C59198.2023.00094

[3] https://github.com/eclipse-glsp/glsp-server-node/pull/127, https://github.com/eclipse-glsp/glsp-client/pull/474

[2] Anthropic. 2024. What is the Model Context Protocol (MCP)? - Model Context Protocol. https://modelcontextprotocol.io/docs/getting-started/intro. [Accessed 13-01-2026].

[3] Carlos Avila, Daniel Ilbay, and David Rivera. 2025. Human–ai teaming in structural analysis: A model context protocol approach for explainable and accurate generative ai. *Buildings* 15, 17 (2025), 3190.

[4] Eranga Bandara, Safdar H Bouk, Ravi Mukkamala, Abdul Rahman, Xueping Liang, Ross Gore, and Sachin Shetty. 2025. Slice-MCP—Fine-Tuned Llama-4 LLM and MCP Enabled 5G Network Slice Orchestration Platform. In *MILCOM 2025-2025 IEEE Military Communications Conference (MILCOM)*. IEEE, 1506–1511.

[5] Eranga Bandara, Sachin Shetty, Ravi Mukkamala, Ross Gore, Abdul Rahman, Peter Foytik, Safdar H Bouk, Xueping Liang, Ng Wee Keong, and Kasun De Zoysa. 2025. Model Context Contracts-MCP-Enabled Framework to Integrate LLMs With Blockchain. In *2025 7th International Conference on Blockchain Computing and Applications (BCCA)*. IEEE, 336–343.

[6] Dominik Bork, Syed Juned Ali, and Ben Roelens. 2023. Conceptual Modeling and Artificial Intelligence: A Systematic Mapping Study. *CoRR* abs/2303.06758 (2023). arXiv:2303.06758 doi:10.48550/ARXIV.2303.06758

[7] Javier Cámara, Javier Troya, Lola Burgueño, and Antonio Vallecillo. 2023. On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML. *Software and Systems Modeling* 22, 3 (2023), 781–793.

[8] Sien Chen, Ruoxian Zhao, Jian-Bo Yang, and Yinghua Huang. 2025. AISStream-MCP: A Real-Time Memory-Augmented Question-Answering System for Maritime Operations. *Journal of Marine Science and Engineering* 13, 9 (2025), 1754.

[9] Juri Di Rocco, Davide Di Ruscio, Claudio Di Sipio, Phuong T Nguyen, and Riccardo Rubei. 2025. On the use of large language models in model-driven engineering. *Software and Systems Modeling* 24, 3 (2025), 923–948.

[10] Eclipse Foundation. [n. d.]. GLSP Documentation - Action Handler. https://eclipse.dev/glsp/documentation/actionhandler/ https://eclipse.dev/glsp/documentation/actionhandler/ (last accessed: 13.03.2026).

[11] Alessio Ferrari, Sallam Abualhaija, and Chetan Arora. 2024. Model generation with LLMs: From requirements to UML sequence diagrams. In *2024 IEEE 32nd International Requirements Engineering Conference Workshops (REW)*. IEEE, 291–300.

[12] Eclipse Foundation. 2026. GLSP Documentation. https://eclipse.dev/glsp/documentation/ https://eclipse.dev/glsp/documentation/ (last accessed: 03.03.2026).

[13] The Linux Foundation. 2025. A2A Protocol. https://a2a-protocol.org/latest/. [Accessed 12-01-2026].

[14] Qize Guo, Yan Chen, Tarik Taleb, and Chao Yang. 2025. Enabling Modular and Intelligent IoRT Systems: Integrating the Model Context Protocol for Semantic Decoupling at the Edge. *IEEE Internet of Things Magazine* 9, 1 (2025), 39–46.

[15] Yunqi Guo, Guanyu Zhu, Kaiwei Liu, and Guoliang Xing. 2025. SensorMCP: A Model Context Protocol Server for Custom Sensor Tool Creation. In *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services*. 747–752.

[16] Andreas Hell. 2026. *Design and Implementation of a Model Context Protocol (MCP) Server for the Graphical Language Server Platform (GLSP)*. Diploma Thesis. Technische Universität Wien. doi:10.34726/hss.2026.137296

[17] Federico Heras. 2023. A Comparison of Enterprise Architecture Tools.. In *ICSBT*. 186–192.

[18] Yoshiki Higuchi, Jee-Eun Kim, and Masahiro Bessho. 2025. MCP-based AI coding agent Integrating Static Analysis for Web Accessibility Error Correction. In *2025 TRON Symposium (TRONSHOW)*. IEEE, 1–4.

[19] Georg Hinkel and Bodo Igler. 2025. An internal DSL for graphical modeling tools based on GLSP. *J. Object Technol.* 24, 2 (2025), 2. doi:10.5381/JOT.2025.24.2.A11

[20] Yi Huang, Bowen Zheng, Yunxi Dong, Hong Tang, Huan Zhao, Rakibul Hasan Shawon, Sensong An, and Hualiang Zhang. 2025. MCP-enabled LLM for meta-optics inverse design: leveraging differentiable solver without LLM expertise. *Nanophotonics* 14, 27 (2025), 5589–5602.

[21] IBM. 2025. Agent Communication Protocol. https://agentcommunicationprotocol.dev/introduction/welcome. [Accessed 12-01-2026].

[22] Ana C Marcén, Antonio Iglesias, Raúl Lapeña, Francisca Pérez, and Carlos Cetina. 2024. A systematic literature review of model-driven engineering using machine learning. *IEEE Transactions on Software Engineering* (2024).

[23] Wataru Matsuda, Mariko Fujimoto, Yoshihiro Hashimoto, Takuho Mitsunaga, and Kenji Watanabe. 2025. Enhancing Incident Response Through Sigma MCP and Operational Policy Integration with LLM. In *2025 IEEE International Conference on Computing (ICOCO)*. IEEE, 30–35.

[24] Lukasz Mazur, Nenad Petrovic, James Pontes Miranda, Ansgar Radermacher, Robert Rasche, and Alois Knoll. 2025. Querying Large Automotive Software Models: Agentic vs. Direct LLM Approaches. *arXiv preprint arXiv:2506.13171* (2025).

[25] Haydar Metin and Dominik Bork. 2023. Introducing bigUML: a flexible open-source GLSP-based web modeling tool for UML. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 40–44.

[26] Haydar Metin and Dominik Bork. 2023. On Developing and Operating GLSP-based Web Modeling Tools: Lessons Learned from BIGUML. In *26th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023, Västerås, Sweden, October 1-6, 2023*. IEEE, 129–139. doi:10.1109/MODELS58315.2023.00031

[27] Haydar Metin and Dominik Bork. 2025. A reference architecture for the development of GLSP-based web modeling tools. *Softw. Syst. Model.* 24, 6 (2025), 1869–1895. doi:10.1007/S10270-024-01257-Y

[28] Manuel Mischak, Charlotte Verbruggen, Philip Langer, and Dominik Bork. 2026. Uncovering LLM's Capabilities in Model-Based Question Answering for UML Class Diagrams. In *Enterprise, Business-Process and Information Systems Modeling*, Han van der Aa, Renata Guizzardi, Simon Hacks, and Luise Pufahl (Eds.). Springer Nature Switzerland, Cham, 331–348.

[29] Parastoo Mohagheghi, Miguel Fernández, Juan Martell, and Mathias Fritzsche. 2008. MDE Adoption in Industry: Challenges and Success Criteria, Vol. 5421. 54–59. doi:10.1007/978-3-642-01648-6_6

[30] Gunter Mussbacher, Benoît Combemale, Jörg Kienzle, Silvia Abrahão, Hyacinth Ali, Nelly Bencomo, Márton Búr, Loli Burgueño, Gregor Engels, Pierre Jeanjean, Jean-Marc Jézéquel, Thomas Kühn, Sébastien Mosser, Houari A. Sahraoui, Eugene Syriani, Dániel Varró, and Martin Weyssow. 2020. Opportunities in intelligent modeling assistance. *Softw. Syst. Model.* 19, 5 (2020), 1045–1053. doi:10.1007/S10270-020-00814-5

[31] Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2024. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems* 37 (2024), 126544–126565.

[32] Tong Qiu, Jingwei Ge, Wenhan Wu, and Fei-Yue Wang. 2025. Agent2Tool: MCP-Enhanced Reliable Integration of LLM Agents and Legacy Optimization Solvers. In *2025 IEEE 25th International Symposium on Computational Intelligence and Informatics (CINTI)*. IEEE, 000623–000628.

[33] Simon Rädler, Luca Berardinelli, Karolin Winter, Abbas Rahimi, and Stefanie Rinderle-Ma. 2025. Bridging MDE and AI: a systematic review of domain-specific languages and model-driven practices in AI software systems engineering. *Softw. Syst. Model.* 24, 2 (2025), 445–469. doi:10.1007/S10270-024-01211-Y

[34] Partha Pratim Ray. 2025. A survey on model context protocol: Architecture, state-of-the-art, challenges and future directions. *Authorea Preprints* (2025).

[35] Juan V. Razvan Radulescu. 2025. Universal Tool Calling Protocol (UTCP). https://www.utcp.io/. [Accessed 12-01-2026].

[36] Joseph Romeo, Marco Raglianti, Nagy Csaba, and Michele Lanza. 2025. UML is back. Or is it?. In *ICSE 2025 47th International Conference on Software Engineering*.

[37] Davide Di Ruscio, Phuong T. Nguyen, and Alfonso Pierantonio. 2023. Machine Learning for Managing Modeling Ecosystems: Techniques, Applications, and a Research Vision. In *Software Ecosystems: Tooling and Analytics*, Tom Mens, Coen De Roover, and Anthony Cleve (Eds.). Springer, 249–279. doi:10.1007/978-3-031-36060-2_10

[38] Donya Sowelem. 2020. WU (Wirtschaftsuniversität Wien) — wu.ac.at. https://www.wu.ac.at/erp/webtrainer/webtrainer-business-process-model-and-notation-bpmn/bpmn-exercies. [Accessed 26-03-2026].

[39] Stefan Szeider. 2025. Bridging language models and symbolic solvers via the model context protocol. In *28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 30–1.

[40] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science* 18, 6 (2024), 186345.

[41] Shutong Wu and Justin P Barnett. 2025. MCP-Unity: Protocol-Driven Framework for Interactive 3D Authoring. In *Proceedings of the SIGGRAPH Asia 2025 Technical Communications*. 1–4.