CityDPC-MCP: Enabling 3D city model processing via Model Context Protocol

Tammo Hamer[1,2] and Maxim Shamovich[1]
[1]Institute of Energy Efficiency and Sustainable Building (E3D), RWTH Aachen University, Germany
[2]E3D Ingenieurgesellschaft mbH, Germany
E-mail(s): tammo.hamer@rwth-aachen.de, shamovich@e3d.rwth-aachen.de

Abstract: 3D city models in CityGML and CityJSON provide an important data foundation for urban analysis, yet they remain difficult to use in practice. Datasets are often large and heterogeneous, attributes may be missing, and many workflows require specialized domain knowledge. Large Language Models (LLMs) can lower this barrier by serving as natural language interfaces, provided that they interact with the underlying data structures reliably and consistently. This paper introduces CityDPC-MCP, a Model Context Protocol (MCP) server that wraps predefined functions for loading, querying, analyzing, and modifying 3D city models within a stateful dataset. The server is evaluated against two configurations available to users today: an LLM operating directly on the provided file and an LLM with code execution in a sandbox. Across five test cases, three models, and 100 repetitions each, the MCP setup achieved higher success rates while keeping the dataset out of the model context, which decouples token consumption from file size and format. An additional exploratory experiment on dataset generation illustrates possible agentic workflows and their current limitations.

Keywords: 3D City Models, Large Language Model, Model Context Protocol



Published in the conference proceedings of the 37. Forum Bauinformatik 2026, Rostock, Germany, DOI: https://doi.org/10.18453/rosdok_id00005849
© 2026 The copyright for this article lies with the authors. Use is permitted under a Creative Commons license Attribution 4.0 International.

1 Introduction

Urban planning, regional analysis, and simulations of energy and heat demand [1] rely on large-scale 3D geospatial data. Standardized formats such as CityGML [2] and CityJSON [3] have improved data availability and interoperability.

Libraries and tools such as CJIO[1], GML Toolbox[2], and citygml4j[3] provide functions to analyze, compute, and modify these models. Although these functions make working with these datasets significantly more convenient, they require programming skills and a deeper understanding of the underlying data structures. This creates a barrier to entry for non-expert users.

Large Language Models (LLMs) can reduce this barrier by serving as natural language interfaces to domain-specific software tools. Related work shows increasing interest in LLM-based geospatial

[1]https://github.com/cityjson/cjio
[2]https://github.com/KIT-IAI/GMLToolbox
[3]https://github.com/citygml4j/citygml-tools

workflows. Kanna and Kolbe proposed frameworks for querying and semantically enriching 3D city models using LLM-based agents [4], [5]. MCP-based approaches have been proposed for interaction with Industry Foundation Classes (IFC) models [6], Urban Building Energy Modeling (UBEM) [7], and retrieval from the CityJSON specification[4]. These approaches target BIM models, energy simulation, or specification retrieval. An interface that exposes analysis and modification functions for semantic 3D city models, and an evaluation of what such tool mediation contributes, are still missing.

Exposing such functions to LLM-based applications requires an interface between the model and the software. Domain functionality can be provided through host-specific scaffolding, for example, bundled instructions, documentation, and scripts loaded into the model context. Such approaches still require the model to generate and execute code against the library, which corresponds to one of the baseline configurations evaluated in this paper. The *Model Context Protocol (MCP)* instead exposes library functions as individually invocable tools with defined input and return schemas, independently of the host application. Its architecture separates the *MCP host*, which represents the AI application, from *MCP clients*, which establish the communication layer, and *MCP servers*, which expose data sources, software functions, and tool interfaces[5].

For a viable interface between users and 3D city models, however, such tool access must be reliable and efficient. Access to domain libraries is increasingly mediated by LLM-based assistants and agentic applications rather than by hand-written scripts [8], [9], which requires that tools are selected correctly, invoked consistently across repeated runs, and return valid results. Unlike the user experience itself, these properties can be measured without user studies. With the future user experience in mind, this paper therefore presents a first step towards natural language access to 3D city models and investigates whether MCP-based tool access can make LLM interaction with them more reliable, efficient, and suitable for agentic workflows, that is, sequences of dependent tool calls that the model plans, executes, and validates without user intervention. To this end, CityDPC-MCP is introduced, an MCP server that exposes selected CityDPC[6] functions for processing CityGML and CityJSON files and supports agentic workflows such as city model generation, scene querying, and semantic enrichment.

2 Methodology

The MCP server implementation is guided by several requirements derived from the objective of supporting the agentic workflows described above. The server should provide the client with a selected set of CityDPC tools that enable both information retrieval and state-changing operations. The selected functions should build on each other to support tool chaining in longer workflows. Both CityGML and CityJSON files should be compatible with the server. The size of the input datasets should not significantly affect the server's functionality. Although the server must handle complex CityDPC output, its return values should remain clearly interpretable by the LLM. CityDPC's structured Python outputs can be exposed through typed interfaces, making them easier for the LLM to interpret. To support appropriate tool selection and invocation for a given instruction, the tool descriptions must

[4] https://github.com/cityjson/cj-mcp
[5] https://modelcontextprotocol.io/docs/learn/architecture
[6] https://github.com/RWTH-E3D/CityDPC

be precise. For maintainability and future updates, the MCP server implementation should remain simple, and the code should be modular to facilitate integrating additional tools.
The implementation approach includes an MCP server that runs MCP tools on a chosen dataset. The server can retrieve a list of datasets from a designated directory and load a dataset from the list as the active dataset. The *Dataset* is a Python object defined in CityDPC that derives its data structure from CityGML and CityJSON files. The system architecture is shown in Figure 1.

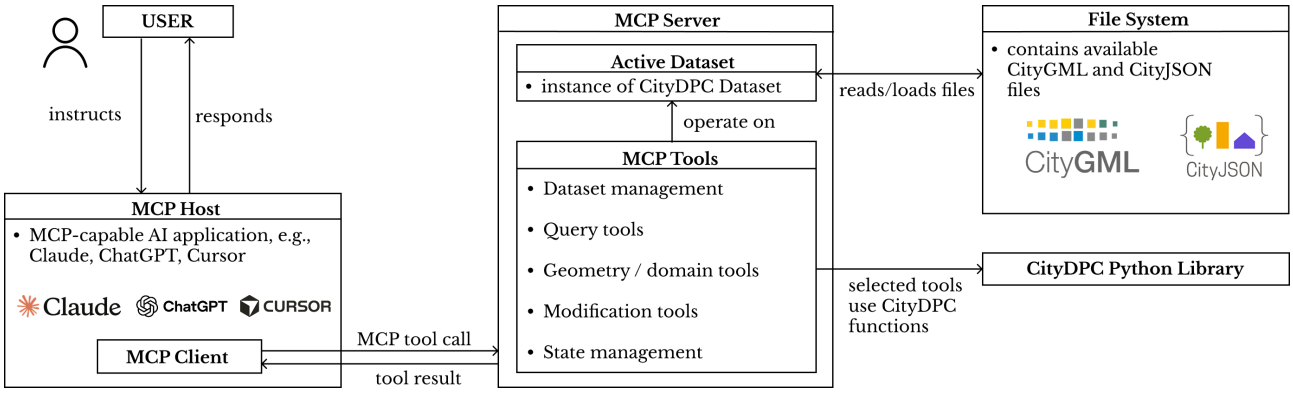


Figure 1: Schematic visualization of the system architecture.

The underlying tool design follows a simple principle. Instead of providing predefined workflows consisting of multiple functions, the MCP server exposes each CityDPC function as an individual tool. For a given instruction, the LLM must therefore decompose the task into individual steps and decide which tools to call and in which order. This design allows different tool chains to be composed for the same task. Dependencies between tools arise from the function parameters required for each call.
The registered tools follow a consistent schema comprising the tool name, description, input parameters, and expected return value. The CityDPC-MCP server contains both CityDPC-imported functions and additional state-management functions, enabling agentic workflows using a git-inspired system. Table 1 shows a structured list of the implemented tools and their purposes. The server implementation, tool definitions, benchmark prompts, and validation scripts are available at https://github.com/HamerCode/CityDPC-MCP.

Table 1: Categories and functions of the CityDPC-MCP tools.

Category	**Example Tools**	**Purpose**
Dataset management	`load`, `save`, `filter`	Stateful dataset handling.
Query tools	`get_IDs`, `get_building`, `analyse`	Controlled information retrieval.
Geometry/domain tools	`roof volume`, `party walls`	Domain-specific computation.
Modification tools	`create`, `enrich`, `remove`	Schema-consistent edits.
State management	`snapshot`, `rollback`	Recovery from faulty edits.

3 Evaluation setup

To survey the extent to which the MCP server improves the ability of an LLM to perform tasks related to 3D city models, the proposed MCP setup is compared with two baseline configurations: (a) an LLM without tool access operating directly on a CityJSON file and (b) an LLM that can write and execute code in a sandboxed environment. For the state-changing test cases, setup (b) receives the file path and modifies the file through generated code, while setup (a) is instructed to return a JSON patch that is applied to the file. The experiment comprises five test cases that simulate representative workflows supported by the MCP server: (1) querying all building IDs, (2) determining the highest building in a dataset, (3) computing the total number of party walls in a dataset, (4) increasing the height of a given building ID, and (5) adding a new building to the dataset.

Success is evaluated per run as a binary criterion; partially correct results are counted as failures. Reference values are computed with CityDPC scripts. For test cases (1) to (3), the model output is parsed into a fixed JSON schema and compared against these values. For the state-changing test cases (4) and (5), correctness is determined from the resulting file rather than from the model output. Each run starts from an identical copy of the input dataset; the modified file is stored separately and the original is restored before the next run. A validation script then checks whether the file is valid CityJSON, whether CityDPC can import it, and whether the requested modification was applied. All checks are performed automatically. Each test case is repeated $n = 100$ times, and the metrics token count and end-to-end time are reported alongside the *success rate*.

The test dataset shown in Figure 2 is a CityJSON dataset in LoD2 containing 36 buildings and originating from the Geoportal Nordrhein-Westfalen[7]. The spatial extent of the original model was substantially reduced to decrease file size and fit the dataset into an LLM's context window.

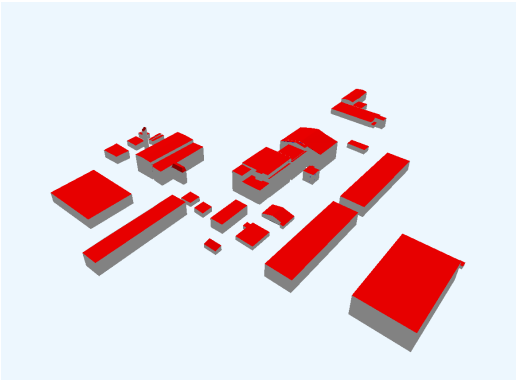


Figure 2: Test dataset.

Three models were used for the evaluation: *gpt-oss-120b*[8](*GPT-OSS*), *Mistral Small 4*[9], and *GPT-5.4 Mini*[10], with all of them configured with high reasoning effort. *GPT-OSS* and *Mistral Small 4* are open-weight models accessed via Inferenz NRW[11], while *GPT-5.4 Mini* was accessed through the OpenAI Responses API as a closed-source reference model. Identical system instructions were used across all models and setups. To complement the controlled benchmark, an additional qualitative campus-planning experiment explored whether the MCP setup can support less deterministic, open-ended agentic workflows. The qualitative experiment was conducted using the Codex App[12] with GPT-5.5 and Claude Code[13] with Claude Opus 4.8. The CityDPC-MCP server was hosted locally on a MacBook Pro (Apple M5, 24 GB unified memory), while the models ran on external servers.

[7] https://geoportal.nrw/
[8] https://openai.com/de-DE/index/introducing-gpt-oss/
[9] https://mistral.ai/news/mistral-small-4/
[10] https://developers.openai.com/api/docs/models/gpt-5.4-mini
[11] https://www.oski.nrw/inference/
[12] Codex App, https://openai.com/index/introducing-the-codex-app
[13] Claude Code, https://code.claude.com/docs/en/quickstart

4 Results

Table 2 reports the setup-level metrics from the evaluation benchmark. Across the tested models, MCP is the most reliable configuration, achieving the highest average success rates while keeping token consumption the lowest. The sandbox setup is flexible in principle, but in the benchmark it often consumed far more tokens and did not reliably solve the geometry-heavy tasks. The no-tool setup relies entirely on the LLM correctly interpreting the file. *Mistral Small 4* did not invoke the code execution tool in any run of the sandbox setup and answered directly from the file content.

The task-level results show that simple attribute queries can often be solved without tools, since the required information is directly available in the file. Tasks involving schema-aware processing, such as counting party walls, demonstrate the advantage of specialized tools. While the MCP server needs only a single tool call to invoke a CityDPC function and interpret its return values, the other approaches had to compute the required values themselves. This resulted in near-zero success rates for the baselines in the geometry-heavy test cases. The state-changing tasks show a similar effect. Increasing an existing attribute requires far less effort than creating a new building compliant with the CityJSON geometry model. The MCP server encapsulates this construction logic in a dedicated tool, while baselines tend to produce incorrect geometry, leading to much lower success rates.

Table 2: Evaluation metrics across setups and models.

		GPT-OSS				Mistral Small 4				GPT-5.4 Mini			
Setup	**Test Case**	**Avg. Tokens**	**Tool Calls**	**Avg. Time(s)**	**Success Rate**	**Avg. Tokens**	**Tool Calls**	**Avg. Time(s)**	**Success Rate**	**Avg. Tokens**	**Tool Calls**	**Avg. Time(s)**	**Success Rate**
MCP	List buildings	10 500	2.03	33.3	85%	13 267	2.88	7.3	98%	9 278	2.05	15.6	100%
	Highest height	17 419	2.09	35.2	100%	22 186	2.03	3.5	94%	14 899	2.32	17.6	99%
	Raise building	16 205	3.92	73.8	88%	14 648	3.27	5.7	8%	21 280	5.28	36.9	100%
	Add building	20 192	5.02	92.9	96%	59 279	5.46	10.2	100%	21 972	4.27	44.1	99%
	Party walls	10 006	2.04	27	98%	19 397	2	3.8	100%	9 716	2	14.7	88%
	Setup avg.	**14 865**	**3.02**	**52.4**	**93%**	**25 755**	**3.13**	**6.1**	**80%**	**15 429**	**3.18**	**25.8**	**97%**
No Tools	List buildings	63 577	0	31.8	1%	81 106	0	8.7	0%	62 405	0	24.2	18%
	Highest height	62 818	0	33.6	96%	80 244	0	1	70%	62 333	0	22.9	100%
	Raise building	60 700	0	21.4	99%	80 136	0	1.7	96%	60 788	0	8.2	99%
	Add building	62 878	0	36.4	5%	81 527	0	24.6	0%	75 408	0	169.3	86%
	Party walls	60 968	0	19.9	0%	80 152	0	0.8	2%	98 461	0	414.6	1%
	Setup avg.	**62 188**	**0**	**28.6**	**40%**	**80 633**	**0**	**7.4**	**34%**	**71 879**	**0**	**127.8**	**61%**
Sandbox	List buildings	219 167	2.45	114.6	44%	82 253	0	11.3	0%	83 146	0.32	28.5	29%
	Highest height	223 802	2.6	107.5	67%	81 448	0	4.4	57%	91 489	0.46	27	100%
	Raise building	226 108	2.63	111	50%	81 994	0	16.5	91%	122 487	1	16.3	99%
	Add building	190 270	2.03	108.7	8%	90 534	0	198.3	2%	138 873	1	185.4	91%
	Party walls	324 671	4.13	162.5	0%	81 357	0	4.1	1%	152 520	1	329	1%
	Setup avg.	**236 803**	**2.77**	**120.9**	**34%**	**83 517**	**0**	**46.9**	**30%**	**117 703**	**0.76**	**117.2**	**64%**
Total	**All Test Cases**	**104 619**	**1.93**	**67.3**	**56%**	**63 302**	**1.04**	**20.1**	**48%**	**68 337**	**1.31**	**90.3**	**74%**

Table 3 reports how the models used the MCP tools[14]. For each test case, a reference workflow defines the tool calls required to solve the task. A tool call is successful if it executes without error and matches a step in this workflow with valid parameters. Workflow adherence is the ratio of successful to actually executed tool calls and therefore measures overhead relative to the reference sequence,

[14] *STC* = Successful Tool Calls (Ø), *ATC* = Actual Tool Calls (Ø), *WA* = Workflow Adherence (%), *SR* = Success Rate (%).

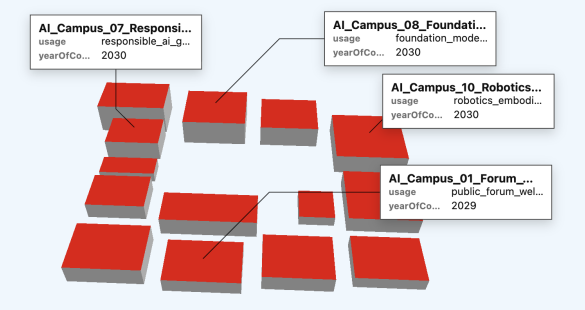
not task performance. *GPT-OSS* and *GPT-5.4 Mini* achieved similar values, while *Mistral Small 4* was considerably lower.

Table 3: Workflow adherence in the MCP setup across the high-reasoning model configurations.

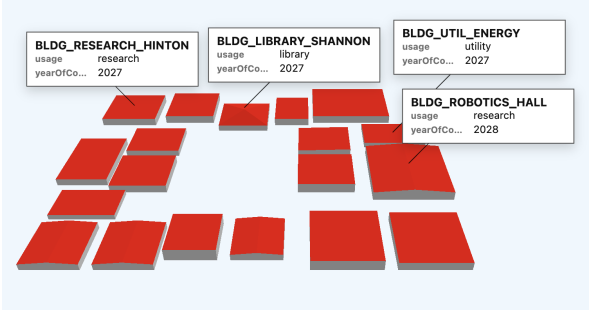
	GPT-OSS				Mistral Small 4				GPT-5.4 Mini			
Test Case	STC (Ø)	ATC (Ø)	WA (%)	SR (%)	STC (Ø)	ATC (Ø)	WA (%)	SR (%)	STC (Ø)	ATC (Ø)	WA (%)	SR (%)
List buildings	2	2.03	98.5	85.0	2	2.88	69.4	98.0	2.03	2.05	99.0	100.0
Highest height	2	2.09	95.7	100.0	2	2.03	98.5	94.0	2.26	2.32	97.4	99.0
Raise building	3.88	3.92	99.0	88.0	1.35	3.27	41.3	8.0	4.52	5.28	85.6	100.0
Add building	3.98	5.02	79.3	96.0	4	5.46	73.3	100.0	3.96	4.27	92.7	99.0
Party walls	1.98	2.04	97.1	98.0	2	2	100.0	100.0	2	2	100.0	88.0
Total	**2.77**	**3.02**	**91.7**	**93.4**	**2.27**	**3.13**	**72.6**	**80.0**	**2.95**	**3.18**	**92.8**	**97.2**

The resulting campus models and the corresponding prompt of the open-ended campus-planning task are shown in Figure 3.

> **Prompt:** You are an urban planner. Your task is to create a new AI campus for a university. You are provided with the CityDPC MCP server. Use the provided tools to create buildings with suitable names, geometries, and dimensions. The goal is to create a concept for a completely new campus area focused on artificial intelligence research. Keep in mind that the buildings should be placed in a sensible manner and that urban planning methods should be applied. Please enrich the buildings with semantic attributes. Continue until you are satisfied with the result. Use `emptyCampus.city.json` as the basis for this.



A) GPT-5.5 via Codex



B) Claude Opus 4.8 via Claude Code

Figure 3: Generated models of an AI campus using the MCP server.

5 Discussion

The results indicate that MCP is most useful when a task requires structured access to domain logic rather than simple information retrieval from a file. Dedicated CityDPC functions raise the success rate for the geometry-heavy test cases from near zero in both baselines to 88% and above in the MCP setup. The context cost becomes even more apparent for CityGML: the same dataset required 60,168 tokens as CityJSON and 169,700 tokens[15] as CityGML. Direct file prompting therefore ties context length to file size and format. For realistic city models with hundreds or thousands of buildings, this quickly exceeds available context windows. The MCP approach addresses this bottleneck by keeping the dataset server-side and providing only the MCP tools as context. While file interpretation limits the baseline setup, tool interpretation remains a challenge for MCP. Effective invocation depends on tool descriptions that accurately describe purpose and usage. One observed failure concerns

[15] https://platform.openai.com/tokenizer

the *enrich_building* tool. Although the function updated selected attributes, its description did not clearly convey its purpose, so it added optional attributes even though they were not requested. However, this issue did not substantially affect the overall results. With average WA values of 91.7% for *GPT-OSS* and 92.8% for *GPT-5.4 Mini*, most tool descriptions were clear enough to support reliable tool invocation. Since all models received identical tool descriptions, the lower value for *Mistral Small 4* cannot be attributed to tool design alone. The *raise building* test case points in the same direction. While *GPT-OSS* and *GPT-5.4 Mini* performed well on this task, *Mistral Small 4* only achieved a success rate of 8%. Instead of first retrieving the original building height and then increasing it by two meters, the model often skipped the required query step and set the building height to two meters directly. This suggests model-specific limitations in following workflows that require planning and multi-step tool usage.

The qualitative workflow experiment illustrates the MCP server's potential to enable agentic workflows. When instructed to create a fictional AI campus (Figure 3), the LLM-based agents autonomously utilized their reasoning capabilities to create, query, and update the AI campus files. The experiment demonstrates technical feasibility. It applies no planning criteria and does not assess the quality of the resulting layout.

The MCP system's performance also depends heavily on the host system. Applications such as Claude Code and Codex add system instructions, conversation steering, and integrated execution loops on top of the model. For agentic workflows, these applications are therefore better suited to MCP deployment than direct API calls.

The generated geometries consist mainly of rectangular buildings with similar orientations. CityDPC currently supports only prism-like LoD2 buildings with selected roof types, so the results do not allow separating this constraint from the limited spatial reasoning of the models [10].

The MCP setup handles larger datasets more effectively than file-context setups, but its performance may still decline as growing function outputs increase context length. The evaluation setup also has limitations. The benchmark uses only 36 buildings and predefined prompts, enabling a controlled comparison but not covering the full scope of the MCP server. The validation confirms schema validity, importability, and the requested modification, but does not assess geometric plausibility, so geometric deficiencies like those observed in the campus experiment would not be captured. End-to-end times depend on both the local system and providers' infrastructure, introducing variability. Reported means should therefore be interpreted as orders of magnitude rather than precise performance differences.

6 Conclusion and outlook

This work asked whether tool-mediated access makes LLM interaction with 3D city models more reliable and efficient. Three findings extend beyond the implementation itself. Tasks that depend on schema or geometry are not solved reliably from file context or generated code, but are solved by exposing the corresponding domain functions; the benefit therefore arises from the functions, not the protocol, which contributes to host independence. Keeping the dataset server-side decouples context length from file size and format, a property of the interface rather than the model. Reliability, however,

is not a property of the server alone: tool descriptions, the selected model, and the host application determine whether it acts as a natural language interface or as a basis for agentic workflows.
Future work should improve tool descriptions, extend the tool suite, and introduce batching or pagination for large tool responses. A functionally equivalent baseline that provides the same functions through native function calling would further separate the contribution of the interface standard from that of the domain logic. CityDPC-MCP could also be combined with additional MCP servers for GIS processing, building-energy simulation, weather data, or regulatory constraints. In such workflows, it would serve as a preprocessing and enrichment component within a broader ecosystem of domain-specific tools.

References

[1] F. Biljecki, J. Stoter, H. Ledoux, S. Zlatanova, and A. Çöltekin, "Applications of 3D City Models: State of the Art Review", *ISPRS International Journal of Geo-Information*, vol. 4, no. 4, pp. 2842–2889, Dec. 2015. DOI: 10.3390/ijgi4042842

[2] T. H. Kolbe, C. Nagel, A. Czerwinski, G. Gröger, and A. Stadler, *OpenGIS City Geography Markup Language (CityGML) Encoding Standard*, 2008. DOI: 10.62973/08-007r1

[3] H. Ledoux, K. Arroyo Ohori, K. Kumar, B. Dukai, A. Labetski, and S. Vitalis, "CityJSON: A compact and easy-to-use encoding of the CityGML data model", *Open Geospatial Data, Software and Standards*, vol. 4, no. 1, p. 4, Jun. 2019. DOI: 10.1186/s40965-019-0064-0

[4] K. Kanna and T. H. Kolbe, "Automatic Enrichment of Semantic 3D City Models using Large Language Models", *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. X-4/W6-2025, pp. 105–112, Sep. 2025. DOI: 10.5194/isprs-annals-X-4-W6-2025-105-2025

[5] K. Kanna and T. H. Kolbe, "Advanced User Interaction with Urban Digital Twins using Large Language Models", *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. X-G-2025, pp. 469–476, Jul. 2025. DOI: 10.5194/isprs-annals-X-G-2025-469-2025

[6] B. K. Nithyanantham et al., *MCP4IFC: IFC-Based Building Design Using Large Language Models*, arXiv:2511.05533 [cs], Oct. 2025. DOI: 10.48550/arXiv.2511.05533

[7] H. Li, Y. Xu, and T. Hong, "EnergyPlus-MCP: A model-context-protocol server for ai-driven building energy modeling", *SoftwareX*, vol. 32, p. 102 367, Dec. 2025. DOI: 10.1016/j.softx.2025.102367

[8] W. X. Zhao et al., *A Survey of Large Language Models*, arXiv:2303.18223 [cs], Mar. 2025. DOI: 10.48550/arXiv.2303.18223

[9] M. A. Ali and F. Dornaika, *Agentic AI: A Comprehensive Survey of Architectures, Applications, and Future Directions*, Oct. 2025. DOI: 10.1007/s10462-025-11422-4

[10] Y. Yamada, Y. Bao, A. K. Lampinen, J. Kasai, and I. Yildirim, *Evaluating Spatial Understanding of Large Language Models*, arXiv:2310.14540 [cs], Apr. 2024. DOI: 10.48550/arXiv.2310.14540