Enabling AI Agents for Semantic 3D City Models through Automated Domain Context Generation

Khaoula Kanna, Thomas H. Kolbe*

Chair of Geoinformatics, Technical University of Munich, 80333 Munich, Germany - (khaoula.kanna, thomas.kolbe)@tum.de

Keywords: Semantic 3D City Models, CityGML, 3DCityDB, Large Language Models (LLM), Model Context Protocol (MCP).

Abstract

Semantic 3D city models offer rich urban information yet remain largely inaccessible to non-expert users due to their complex schemas, deep class hierarchies, and heterogeneous data structures. Large Language Models (LLMs) have demonstrated remarkable capabilities in natural language understanding and code generation. But applying them to domain-specific databases requires contextual knowledge that far exceeds what can be manually written and maintained. This paper presents a framework that automatically generates domain context from CityGML data stored in 3DCityDB instances. The framework decomposes domain knowledge into static components (database schema, query patterns, spatial capabilities) and dynamic components (available object classes, properties and generic attribute classifications). These components are assembled into a structured context representation served via the Model Context Protocol (MCP), forming a shared knowledge layer that most AI agents can consume. We evaluate the framework against five CityGML datasets from different countries where attribute values are given in different languages, levels of detail, and thematic modules. Results demonstrate that the system enables an LLM-based agent to correctly formulate complex SQL queries (including 3D spatial operations, multi-level feature relationships, and nested property access). Beyond querying, the generated context supports the creation of additional domain-specific agents for tasks such as semantic enrichment, data quality assessment, and urban scenario analysis, making the MCP Server a reusable domain knowledge interface for semantic 3D city models.

1. Introduction

Semantic 3D city models have become the backbone for urban planning, environmental simulations, energy calculations, and smart decision making. Standards such as CityGML (Gröger et al., 2012) and IFC (ISO, 2024) provide a rich, multi-scale representation of the urban environment. CityGML represents not only geometry but also semantic relationships between buildings, transportation networks, vegetation, and city furniture across multiple levels of detail. Database implementations like 3DCityDB (Yao et al., 2018) enable the storage and management of these models at city scale, supporting complex spatial queries and thematic analysis. Yet despite their richness, semantic 3D city models remain largely used only by domain experts who possess both the technical skills to navigate complex database schemas and the domain knowledge to interpret deeply nested class hierarchies, coded attribute values, and multi-level feature relationships.

Recent research on Large Language Models (LLMs) have demonstrated remarkable capabilities across the built environment domain. In Building Information Modeling (BIM), LLMs have been applied to retrieve material information from heterogeneous data sources including BIM models and PDFs (Chen et al., 2024) to automatically generate building models from textual descriptions (Du et al., 2026), and to create building energy models for performance simulation (Jiang et al., 2024). In the geospatial domain, (Li & Ning, 2023) developed LLM-Geo, an autonomous GIS agent that generates analytical workflows using GPT-4, while (Zhang et al., 2024) proposed GeoGPT for solving spatial analysis tasks through LLM-driven tool invocation. These works share a common pattern: an LLM interprets user intent and generates executable operations (code, queries, or model specifications) against domain-specific data. However, in each case, the connection between the LLM and the data model was established through manual prompt design and hardcoded domain knowledge. This approach works but does not generalize. Because when the data model changes, the prompts must be rewritten.

A naive solution would be to include the entire database schema and domain documentation in the agent's prompt. This approach fails on two sides. First, CityGML and related standards like IFC define hundreds of object classes with deep inheritance hierarchies and thousands of properties, exceeding the context window limitations of current LLMs. A single 3DCityDB instance may contain only a small subset of these classes, yet the full schema documentation would consume the agent's entire available context. Second, much of the relevant knowledge is not static but data dependent. For example, which object classes are present, which properties carry values, which levels of detail are available, which spatial queries are allowed and what generic attributes are defined. This knowledge can only be determined by inspecting the database at runtime. These observations lead to two research questions that guide this work:

1. How can the domain knowledge embedded in semantic 3D city models be made automatically accessible to AI agents?
2. Given the complexity of standards like CityGML and IFC, with their complex data models, how can the relevant data structures be dynamically selected and assembled into a concise, machine-readable context that fits within the token limit of current LLMs?

This paper addresses these questions by presenting a framework for automated domain context generation for semantic 3D city models. The key observation is that 3DCityDB v5 (Yao et al., 2025) does not merely store CityGML data, but it also stores the CityGML data model itself within its schema. Our framework exploits this by scanning the database. The resolved domain knowledge is assembled into a structured context representation and served to AI agents via the Model Context Protocol (MCP). The framework operates on the database representation, rather on raw CityGML/CityJSON files, enabling compact, dataset-specific contexts whose size depends on the schema rather than the number of features.

2. Related Work

The intersection of artificial intelligence and geospatial science, often called GeoAI, has seen rapid growth in the last years, especially with the emergence of LLMs. Recent works such as LLM-Geo (Li & Ning, 2023), GeoGPT (Zhang et al., 2024) and ChatGeoAI (Mansourian & Oucheikh, 2024) showed that LLMs

This contribution has been peer-reviewed. The double-blind peer-review was conducted on the basis of the full paper.
https://doi.org/10.5194/isprs-annals-XII-4-W1-2026-211-2026 | © Author(s) 2026. CC BY 4.0 License.

can be used to lower the barrier between non-expert users and complex geospatial systems. However, they primarily target 2D vector/raster data and general GIS operations. None address the specific challenges of semantic 3D city models with their deep class hierarchies, multi-level-of-detail representations, and the complex relational structures of standards like CityGML (Kutzner et al., 2020).

The 3D City Database (3DCityDB) is the most widely adopted open-source solution for storing and managing CityGML data (Yao et al., 2018). The recently released version 5 introduces a streamlined schema based on CityGML 3.0, reducing the number of tables from 66 to 17 through a more generic Entity-Attribute-Value (EAV) mapping approach (Yao et al., 2025). While this redesign simplifies schema maintenance and extensibility, it also increases the complexity of query generation. Instead of retrieving information from dedicated thematic tables, applications must navigate hierarchical property structures, namespace identifiers, and parent-child relationships to access even basic building attributes. Consequently, generating correct database queries requires detailed knowledge of both the CityGML data model and the database schema.

Recent work has begun addressing this accessibility gap through different approaches. Kanna et al. presented a framework employing LLMs to interact with an Urban Digital Twin, using CityGML for 3D city model representation and the SensorThings API for dynamic sensor data (Kanna and Kolbe, 2025). The framework demonstrated that LLMs can translate natural language queries from diverse stakeholders into complex geospatial and temporal operations. However, the approach relied on manually prepared domain context and was designed for the 3DCityDB v4 schema, limiting its transferability to other datasets without prompt re-engineering. Liu et al., addressed the same problem by transforming CityGML data into a knowledge graph and employing a multi-agent chatbot to generate graph queries. Although this improves reasoning over semantic relationships, it requires an additional data transformation process and the deployment of separate graph database infrastructure (Liu and Wang, 2025).

These approaches represent domain-specific instances of the broader Text-to-SQL problem, in which natural language requests are translated into executable database queries. A critical sub-task identified across these approaches is schema linking, which means mapping natural language references to the correct database tables and columns (Liu et al., 2025). This challenge becomes even more pronounced for semantic 3D city model databases because their hierarchical object classes and generic property representations provide less explicit schema information than conventional relational databases.

The evolution from simple LLM prompting to agentic architectures, where models iteratively plan, invoke tools, observe results, and refine their approach, has enabled increasingly complex task completion (Schick et al., 2023; Yao et al., 2023). Frameworks such as LangChain (Chase, 2022) and LlamaIndex (Liu, 2022) provide infrastructures for building tool-augmented agents that can execute code, query databases, and interact with external APIs. However, integrating new databases or domain-specific services into these frameworks typically requires custom implementations for every application, resulting in limited interoperability and poor reusability.

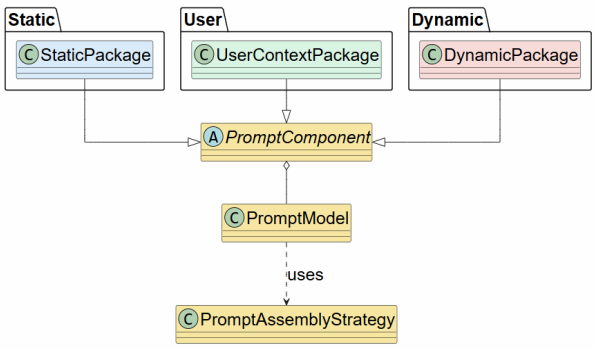

Figure 1. Overall framework architecture of the framework (simplified for visibility reasons)

To address this limitation, Anthropic introduced the Model Context Protocol (MCP) in late 2024 as an open standard for connecting AI agents with external tools and data sources. MCP defines a standardized client-server architecture in which servers expose tools, resources, and prompts through a common JSON-RPC interface (MCP Specification, 2025). While MCP has rapidly gained adoption as a general interoperability standard for AI applications, its potential for exposing the rich semantics of complex geospatial databases has not yet been investigated.

This paper addresses this gap by proposing an MCP-based framework that exposes domain-specific knowledge and geospatial functionality for semantic 3D city models. Rather than relying on manually engineered prompts, dataset transformations, or application-specific integrations, the proposed approach enables AI agents to interact with CityGML-based urban databases through a standardized interface, supporting natural language querying as well as more advanced urban analysis workflows.

3. Methodology

The framework treats 3DCityDB v5 as both a data store and a metadata source. Although LLMs may have prior knowledge of the CityGML conceptual model, they cannot know how CityGML is mapped in the 3DCityDB, such as which classes and properties are actually present or which generic attributes and coded values are used. Storing the schema alongside the instance data enables the framework to combine schema-driven discovery with validation against the actual database contents. Moreover, the database allows semantic and spatial queries at the same time. The challenge lies in the scale and variability of this knowledge: a full CityGML schema defines hundreds of classes and thousands of properties, yet any given dataset instantiates only a subset, with data-dependent generic attributes and spatial configurations. Our framework addresses this by treating the database as both data store and metadata source, automatically scanning its structure at runtime to generate a machine-readable domain context specific to the dataset it contains.

Figure 1 presents the framework architecture. At its core, the *PromptModel* aggregates multiple *PromptComponent* instances, each representing a distinct unit of domain knowledge, and delegates to a *PromptAssemblyStrategy* for token-budgeted composition. Components are organized into three packages:
- **Static Package**: Components whose content is determined by the 3DCityDB relational schema and query guidelines. Static components are written once per schema version and reused across all datasets.
- **Dynamic Package**: Components whose content is discovered at runtime through database introspection (which means querying the database's own metadata tables such as objectclass, property definitions to discover what structures are present, rather than relying on external schema files or manual configuration). Furthermore, instance data is examined to determine present object types, properties and property values. Also, few shot examples for SQL queries are added.

- **User Context Package**: Components that reflect the current interaction session, such as user role, language preferences, and history context.

Each *PromptComponent* declares a priority, token count, and dependency list. The abstract *PromptComponent* class enables components to be selectively activated based on the current dataset and user session. This model-driven architecture (MDA)-based approach ensures that the assembled context contains only relevant information and remains within the token budget for the maximum context length of the target LLM.

3.1 Static Context Components

Static components describe the general static knowledge that derives from the 3DCityDB v5 schema definition, as well as generic query guidelines. First, the *DatabaseSchema* describes the 3DCityDB v5 table definitions, including primary keys, foreign key relationships, column types, and nullability constraints. This compact representation reduces the schema from several thousand characters of SQL to approximately 2,500 characters while preserving all information required for SQL generation. Second, the *QueryGuidelines* component describes the query patterns for the 3DCityDB v5 EAV model. For example, hierarchical property access via parent_id self-joins for nested properties such as the building height, building-to-surface linking through val_relation_type, 3D volume computation via CG_Volume() with ST_IsClosed() validation, and objectclass-aware filtering using objectclass_id. Rules are assigned severity levels to indicate their importance for correct query formulation. Note, that the agent does not interpret geometry directly, it generates SQL that delegates geometric computation to PostGIS and SFCGAL functions on the database, guided by the spatial function reference in the assembled context.

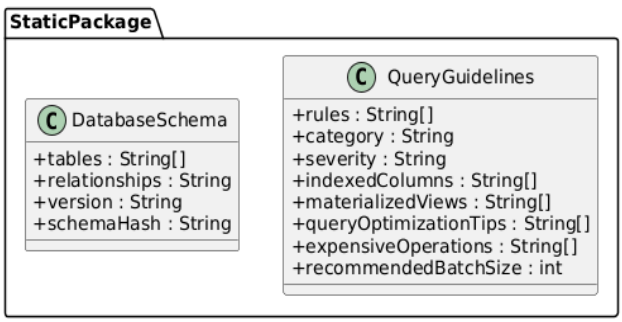

Figure 2. Static Package components overview

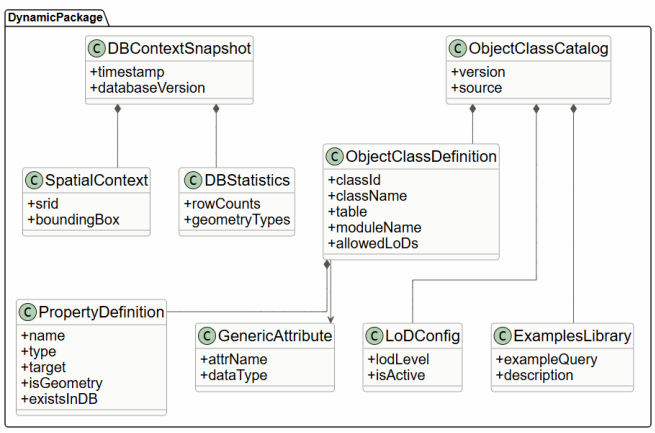

Figure 3. Dynamic Package components overview (simplified for visibility reasons, CodeListDefinition and CodeEntry classes are removed here)

3.2 Dynamic Context Components

Dynamic components are discovered at runtime through database introspection. This section details the four discovery mechanisms: object class scanning, property resolution, codelist mapping, and generic attribute classification. These classes are further visualized in Figure 3.

3.2.1 Object Class discovery and hierarchy resolution: The dynamic introspection begins by querying the feature table to identify all objectclass_id values that exist in the feature table. These IDs are resolved against the objectclass table to retrieve class metadata (classname, namespace assignment, superclass reference, and the embedded JSON schema definition that 3DCityDB v5 stores for each class and which contains the CityGML data model for the class). Only top-level classes are included in the primary catalog, as non-top-level classes such as boundary surfaces are accessed through their parent features rather than queried independently. For each discovered object class, the framework resolves the full inheritance chain by walking the superclass_id references upward through the class hierarchy. An example of the JSON schema can be seen in Figure 4. The names, types and definitions in each property of the CityGML feature are literally copied from the feature catalogue in the CityGML 3.0 standard. At each level, the framework parses the JSON schema to extract the property definitions declared at that level and records the corresponding namespace_id. The resulting *ObjectClassCatalog,* thus, contains not only the instantiated classes but their full inheritance context, forming the input for property resolution (next section). The catalog is constrained by an OCL rule ensuring that only database-validated classes are included: every class must correspond to an objectclass_id present in the feature table, and the hierarchy must be fully resolved. Moreover, based on the resulting *ObjectClassCatalog*, examples of SQL queries will be added to the prompt as few shot examples. In fact, we have created an *ExampleLibrary* for different SQL queries scenarios, including the different LoDs and the different CityGML classes.

3.2.2 Property Resolution: While the *ObjectClassCatalog* provides the CityGML schema-defined properties collected across the inheritance chain, not all these properties necessarily carry values in a given dataset. A CityGML schema may define dozens of properties for a Building, such as yearOfDemolition and storeysBelowGround that are absent in a particular dataset. Including absent properties in the context wastes tokens and risks misleading the LLM into constructing queries that return empty results.
The property resolution step therefore validates the CityGML schema-defined properties against the actual database contents. For each object class, the framework queries the property table for all distinct properties that exist for features of that class. Only surviving properties, meaning those present in both the schema and the data, are kept.

For each surviving property, the framework determines the access path based on its datatype. Scalar types (Integer, Double, String, Timestamp) map directly to value columns in the property table (val_int, val_double, val_string, val_timestamp). Complex types map to join relationships: GeometryProperty joins via val_geometry_id to the geometry_data table, FeatureProperty joins via val_feature_id to the feature table for boundary surfaces and sub-features, and AddressProperty joins via val_address_id to the address table. For properties of type core:Code, the

This contribution has been peer-reviewed. The double-blind peer-review was conducted on the basis of the full paper.
https://doi.org/10.5194/isprs-annals-XII-4-W1-2026-211-2026 | © Author(s) 2026. CC BY 4.0 License.

framework additionally resolves codelist mappings as detailed in Section 3.2.3.

At the end we have for each object class, a list of *PropertyDefinition* instances that tell the LLM exactly how to access each attribute: the property name to filter on, the namespace ID, which value column or join table to use, and for coded properties, what the code values mean.

3.2.3 Codelist Resolution Strategy: CityGML properties of type core:Code store enumerated values as code strings (e.g., "1000" for a building function). Without code mapping or code resolution, the LLM cannot translate a user's question about "residential buildings" into the correct code filter. Therefore, the framework searches the codelist and codelist_entry tables in 3DCityDB for a matching codelist, using substring and camelCase-split matching between the property name and codelist identifiers. When a match is found, the stored code definitions are retrieved for the specific code values present in the dataset.
However, if the codelist Table is empty or doesn't contain the codelist of the target property (e.g. usage or function), the distinct code values are included as they are.

3.2.4 Generic Attribute Classification: CityGML allows datasets to carry application-specific attributes beyond the standard schema, stored in 3DCityDB v5 as properties with the namespace_id = 3. These generic attributes vary dramatically and linguistically across datasets. The framework classifies each generic attribute by querying its value distribution. String and numeric attributes with a distinct value count below a configurable threshold (default is 20 – can be configured) are classified as categorical and their complete value lists are included, enabling the agent to suggest valid filter values. Numeric attributes above the threshold include only their value range (min/max). String attributes above the threshold include sample values to give the agent a sense of the data without exhaustive enumeration.

To manage token consumption when datasets carry large numbers of generic attributes, the framework applies a three-step filtering pipeline: (1) constant removal : attributes with only one distinct value carry no discriminative information and are excluded; (2) deduplication : case-insensitive duplicate names are merged; (3) prefix-based grouping : when the attribute count exceeds a configurable threshold (default: 25 – can be configured), attributes sharing a common prefix (identified dynamically by splitting on underscore delimiters) are collapsed into summary entries that report the group size and the five most variable sub-attributes.


```
"identifier": "bldg:AbstractBuilding",
"description": "AbstractBuilding is an abstract superclass representing the
common attributes and associations of the classes Building and BuildingPart.",
"table": "feature",
"properties": [
    {
        "name": "class",
        "namespace": "http://3dcitydb.org/3dcitydb/building/5.0",
        "description": "Indicates the specific type of the Building or
        BuildingPart.",
        "type": "core:Code"
    },
    {
        "name": "function",
        "namespace": "http://3dcitydb.org/3dcitydb/building/5.0",
        "description": "Specifies the intended purposes of the Building or
        BuildingPart.",
        "type": "core:Code"
    },
```


Figure 4. A small snapshot of the embedded JSON Schema in the Objectclass Table for the AbstractBuilding Class

3.3 Context Assembly

The assembly process composes the static and dynamic components into a single structured text representation. The *PromptAssemblyStrategy* assembles the generated components (schema, guidelines, object class catalog, generic attributes, 1-shot example) while enforcing a configurable token budget. Optional components (more examples, user history) are included in priority order until the context budget is exhausted. If the dynamic components exceed the budget (possible with datasets containing many object classes or extensive generic attributes) lower-priority components are shortened or removed. The assembly validates the final token count against the budget and ensures all required components are present, as specified by the OCL rule *TokenBudgetNotExceeded*.

In the Dynamic Package, the few shot examples come from the *ExamplesLibrary,* which depends on the *ObjectClassCatalog* (to filter examples by available classes), and the *LoDConfig* constrains which geometry properties appear in the catalog.

Then, the assembled context is exposed through the Model Context Protocol as an MCP prompt resource. When an MCP client connects, the server executes the full discovery and assembly pipeline, generating a context according to the current database state. The context is cached with a configurable freshness window (tracked via the timestamp and maxAgeMinutes fields of *DBContextSnapshot* class of the Dynamic Package) and regenerated when the cache expires or when the client explicitly requests a refresh. In addition to the assembly tool, the MCP server exposes individual tools for executing SQL queries against the database and for highlighting objects in a 3D Viewer, allowing the LLM agent to iteratively query and refine its understanding.

This logic of separating between static and dynamic components, in addition to managing the user context, can be replicated and reused with other standards and databases, such as IFC. The generated context functions as a machine-readable dataset description. Any MCP-compatible agent can consume it, regardless of its task.

4. Implementation

The framework is implemented as a Python-based MCP server that connects to 3DCityDB v5 instances running on PostgreSQL with PostGIS extension. This section describes the server architecture, the query agent built as a primary client, and the 3D visualization component. The developed MCP server is available under this link: https://github.com/tum-gis/3dcitydb-mcp-server.

4.1 MCP Server Architecture

The MCP server is implemented using the official Python MCP SDK and exposes both prompt resources and tools through the standardized JSON-RPC interface. Upon client connection, the server executes the full context generation pipeline described in Section 3. It connects to the configured 3DCityDB instance, performs object class discovery, property resolution, codelist mapping, generic attribute classification, and few-shot examples, then assembles the result into a structured domain context. This context is served as prompt that the connecting agent receives as part of its system instructions upon calling the MCP assembly tool. The duration of the prompt setup ranges from a couple of seconds to 5 minutes depending on the complexity of the data stored in the database.

This contribution has been peer-reviewed. The double-blind peer-review was conducted on the basis of the full paper.
https://doi.org/10.5194/isprs-annals-XII-4-W1-2026-211-2026 | © Author(s) 2026. CC BY 4.0 License.

In addition to the assembly tool, the MCP server exposes a set of tools that agents can invoke during their reasoning process. One primary tool is run_query, which accepts a SQL string, executes it against the database with read-only permissions, and returns the result set.

The server supports two transport modes defined by the MCP specifications: stdio for local deployment, where the server process communicates with the client through standard input/output streams, and SSE (Server-Sent Events) for remote deployment over HTTP. A Docker image is provided that bundles the server with its dependencies and accepts database connection parameters as environment variables, enabling deployment alongside existing 3DCityDB Docker containers.

4.2 Query Agent

To evaluate the framework's effectiveness, we implemented a natural language query agent as the primary MCP client. The agent is built using LangChain and follows the ReAct framework (Yao et al., 2023). Given a user question, the LLM reasons about the required query structure, generates a SQL statement, invokes the run_query tool, observes the result, and either returns the answer or refines the query based on error messages or unexpected results. The agent does not need separate tools for schema exploration because this information is already embedded in its system prompt. The LLM model is configurable, as it can be used with OpenAI, Anthropic or with Ollama models. Our experiments use mainly Ollama to evaluate the framework across different locally running LLMs.

The MCP server reports the token count of the assembled domain context alongside the assembly tool, enabling the agent to allocate its remaining context window accordingly (we opted for 32768 for performance reasons). The domain context remains fixed throughout the session, while older conversations are progressively discarded as the dialogue grows.

4.3 Visualization and User Interaction

Beyond query generation, the MCP server exposes a highlight_features tool that accepts a list of feature identifiers and a visual encoding (color, opacity) and emits a highlight event to any connected viewer. This tool is viewer-agnostic by design, meaning it can be consumed by Cesium-based engines, the 3DCityDB Web-Map-Client, or any custom viewer. When the agent executes a spatial query such as "which buildings are within 200 meters of the school?", it can invoke highlight_features with the resulting object identifiers, and the connected viewer renders the selection without the agent needing to know which viewer is active. This reinforces the MCP server's role as a shared knowledge and interaction layer between agents and heterogeneous client applications.

To support use cases that go beyond passive visualization, such as urban scenario analysis and what-if simulations, we additionally developed a custom 3D viewer built on Three.js (Figure 5). Unlike read-only viewers, this viewer allows users to modify geometry directly within the scene by extruding buildings to simulate height changes, replacing roof shapes to evaluate solar potential under different roof types, or inserting new vegetation objects alongside a road. The viewer loads CityGML geometry from the 3DCityDB, supports thematic coloring based on query-derived attributes, and provides standard 3D navigation. While the visualization and geometry editing components are not the primary focus of this paper, they illustrate how the domain context served by the MCP server enables a range of urban applications beyond natural language querying.

Moreover, to validate the framework's reusability, we additionally implemented a data quality agent that connects to the same MCP server. This is discussed in Section 6.

5. Evaluation

We evaluated the framework against five CityGML datasets from three different countries, three languages, and multiple levels of detail. First, the Munich LoD2 dataset represents a residential district in Munich with buildings at LoD 2, using German ALKIS codelists. The Tokyo dataset contains texturized LoD2 buildings. The Ingolstadt dataset is evaluated in two configurations: LoD2 and LoD3 buildings with transportation infrastructure, including road sections, intersections, traffic spaces, vegetation, and city furniture. The Netherlands dataset provides Dutch buildings from Rotterdam at LoD 2.

Given that each LLM calculates tokens differently, even for Ollama models, we have considered 4 characters per token for a generalization. This doesn't work specifically for any large language models. It's an approximation that happens to be reasonable for English text with GPT-style tokenizers but can be off by 20-40% for other models.

Figure 6 presents the token composition of the optimized context generated for each dataset. Static components (database schema and query guidelines) remain constant across all datasets at 2118 tokens. Dynamic components vary according to the data present in each instance, ranging from 2,446 tokens for Tokyo (single module, minimal generic attributes) to 4,055 tokens for Ingolstadt LoD3 (five CityGML modules, 17 non-toplevel

Figure 5. Interaction with the LLM agent and a Threejs based 3D Viewer using the MCP Server

classes, extensive generic attributes). Moreover, as can be spotted in the same figure, moving from LoD2 to LoD3 nearly doubles the number of tokens in the dynamic context (from 2532 to 4055) due to the addition of transportation classes. However, the total context increase is only 33% because the static components absorb none of the increase, which shows the efficiency of the framework.

To quantify the framework's optimization, we compare the optimized context against an unoptimized baseline that includes all CityGML object classes (not just instantiated ones), all schema-defined properties (without database validation), all generic attributes (without constant removal, deduplication, or prefix grouping), and a verbose schema rendering (Figure 7). In fact, the unoptimized baseline for most datasets ranges from 388k to 1.97M tokens, exceeding the context window of most current LLM. The Ingolstadt LoD3 unoptimized figure (1.97M tokens) is notably higher than the other datasets because its unfiltered generic attribute set, containing the full OpenDRIVE attributes without grouping, accounts for 388k tokens alone in addition to LoD3 building features such as doors, windows and building installations. The optimized version compresses this to 2,532 dynamic tokens.

5.1 Query Evaluation

5.1.1 Benchmark Design: The query agent is evaluated against a benchmark of 100 natural language queries of three query types: semantic (property access, codelist resolution, aggregation), geometric (distance, volume and area calculations), and topological (adjacency, connectivity). The queries are organized into four complexity levels based on the SQL structure required for the ground truth answer. Simple queries (n=18) involve single-table lookups and feature counts. Medium queries (n=33) require property joins with namespace filtering, codelist resolution, or basic aggregation. High-complexity queries (n=34) involve boundary surface relationships, 3D geometry computations, or spatial topology operations. Very high-complexity queries (n=15) require multi-level relationship chains, spatial joins, or arithmetic operations combining multiple geometric computations. An example of a very high complex query is "*Which residential buildings have a window-to-wall ratio above 0.3*?". The complete query list and complexity classification is available under this link: https://handy-porpoise-645.notion.site/Evaluation-Questions-31b3fb8cb4358084bfcdc6b4023c4c15

The query agent was designed using LangChain and Ollama. The agent connects to the MCP server and calls the assemble_prompt() function to get the context of the database. Then, the agent gets access to the evaluation questions. Each question is evaluated independently with an empty conversation history to ensure reproducibility. The questions were not repeated. In fact, all evaluations use greedy decoding (temperature 0), ensuring deterministic outputs, which means repeated questions yield identical answers. To evaluate the output, four metrics are used. (1) Execution Accuracy (EX): this metric measure how accurately a generated SQL query retrieves the expected results from a database. For this we have manually created the ground truth SQL queries for each question and compared them with the agent generated SQL queries. (2) The SQL Validity Rate: this metric measures the percentage of queries for which the agent produces executable SQL after up to five attempts, capturing iterative self-correction through agent–database interaction. (3) First-Attempt Success Rate: this measures the percentage of queries where the first generated SQL executes successfully, indicating how well the context guides the agent without trial-and-error. And (4), Median Latency captures the time the agent took from question to answer in seconds. All experiments of the query agent were conducted on a Lenovo ThinkPad X1 Carbon Gen 12 (Intel Core i7) laptop with 32 GB RAM, operating on Windows 11 Pro. The LLMs were installed via Ollama on a dedicated GPU server with 128 GB RAM, an NVIDIA RTX 6000 Ada Generation GPU with 48 GB GDDR6 VRAM, and two CPUs (48 cores, 2.3–3.3 GHz) running Ubuntu. The 3DCityDB instance was hosted on a virtual machine with 24 GB RAM, 2 CPUs, no GPU, and Ubuntu 22.04.5 LTS.

5.1.2 Cross-Dataset Results: Table 1 presents the query agent's performance across all five datasets using Qwen 3.5 27B with 4-bit quantization. The table evaluates whether the dynamically generated context generalizes across datasets with different characteristics. If the framework works correctly, SQL validity should remain high regardless of the dataset's country of origin, language, or CityGML classes. The most interesting finding is the consistency of SQL validity across datasets: all five datasets achieve between 92.6% and 95.7%, despite significant differences in country of origin, generic attribute and properties language, and schema complexity. This confirms that the framework's dynamic context generation produces effective prompts regardless of the specific dataset characteristics. For example, asking the agent "show me all residential buildings" was successfully executed by the agent for all the five datasets with attribute data in three different languages. The agent was able to resolve the codelist for each dataset and retrieve the residential buildings accordingly.

Execution accuracy (EX) ranges from 61.2% (Tokyo LoD2) to 76.7% (Munich LoD2). The EX for Ingolstadt LoD3 was 69.9% which is not bad even though the dataset covers five CityGML modules (Transportation, Construction, Building, City furniture and Vegetation). This can be attributed to the richer context generated for this dataset which provides the agent with more

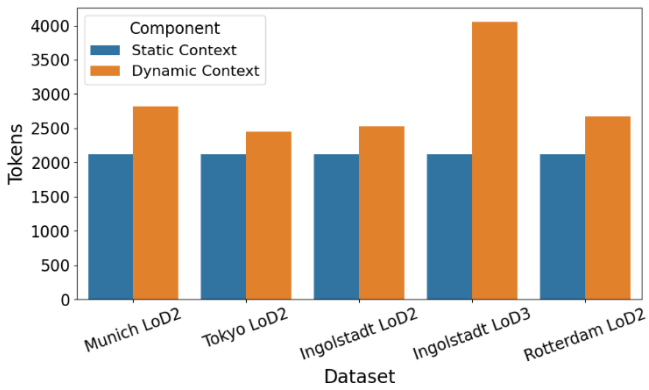

Figure 6. Number of token distribution per dataset

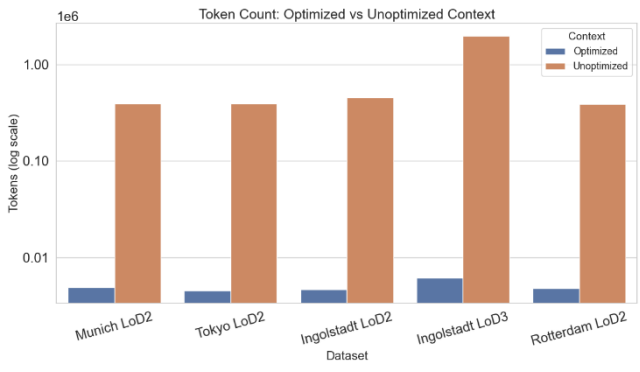

Figure 7. Token count for optimized vs unoptimized context

structural guidance for complex queries. The Tokyo dataset shows the lowest first-attempt success rate (46.9%) and highest average retries (2.5), suggesting that cross-language operation introduces additional complexity. The agent occasionally generates queries using English property names that need to be corrected to match the Japanese attribute labels stored in the database. However, the SQL validity remains high (93.3%), indicating that the agent corrects itself effectively.

	MUC	I2	I3	NL	JP
EX (%)	76.7	71.4	69.9	64.9	61.2
SQL Validity (%)	95.1	93.5	95.7	92.6	93.3
First-Attempt Success (%)	60.8	59.1	53.0	54.9	46.9
Avg Retries / Query	1.9	2.2	2.1	2.2	2.5
Median Latency (s)	8.2	8.8	10	12.1	14.1

Table 1.Cross-dataset evaluation (Qwen 3.5 27B)

Complexity	Metric	MUC	I2	I3	NL	JP
Simple	**EX**	92	83	81	74	73
	SQL Valid	100	82	88	82	82
Medium	**EX**	87	74	74	67	69
	SQL Valid	100	100	98	97	96
High	**EX**	69	57	74	53	53
	SQL Valid	97	95	98	94	98
Very High	**EX**	56	50	47	62	50
	SQL Valid	100	98	97	98	95

Table 2. Breakdown by Complexity (in % by the Qwen 3.5 27B model)

Moreover, the complexity breakdown as can be shown in Table 2 reveals a consistent pattern across all datasets: SQL validity remains above 94% at every complexity level except Simple, where queries answerable directly from the context (without SQL) lower the metric. Execution accuracy shows the expected degradation from Simple to Very High, with the steepest drop occurring at the Very High level for Ingolstadt LoD3 (47%). These are the multi-level transportation queries (Intersection to TrafficSpace then to TrafficArea) that represent the most challenging query patterns in the benchmark.

Importantly, zero errors were recorded across all query executions, meaning the agent never crashed, timed out, or produced unrecoverable failures. Every query received an answer, even when the SQL required multiple retries.

5.1.3 Cross-Model Results: Table 3 presents the cross-model comparison across four LLMs ranging from 14B to 120B parameters, all evaluated on the Munich LoD2 dataset using the assembled context. The results reveal that the framework enables effective querying across a wide range of LLM model sizes, with execution accuracy between 70.6% and 79.5% and SQL validity between 87.3% and 95.1%. This represents a remarkably small difference given the parameter difference between the smallest and largest model.

The most notable finding is the absence of a clear correlation between model size and execution accuracy. Qwen 2.5 (32B) achieves the highest overall EX at 79.5%, outperforming the substantially larger GPT-OSS (120B) at 70.6%. The smallest model, Ministral 3 (14B), matches GPT-OSS at 70.6% EX despite having less than one-eighth of the parameters. This suggests that, given sufficient domain context, the quality of the assembled prompt matters more than model capacity for this task.

The models exhibit distinct behavioral profiles. GPT-OSS (120B) achieves the highest first-attempt success rate (68.6%) and lowest retry count (1.4), indicating that the larger model more often generates correct SQL on the first try. In fact, it struggles with Simple queries (55% EX – meaning out of the 18 simple queries, the agent got 10 out of them. So roughly 8 queries failed, meaning the agent either produced wrong SQL or answered from the context without producing an SQL query) where smaller models perform better, because it sometimes over-engineers straightforward COUNT queries. Ministral 3 (14B), on the other hand, shows the lowest first-attempt success (41.2%) and highest retry count (2.6) but achieves competitive accuracy through self-correction. This generates more SQL attempts but eventually achieves correct answers.

	Ministral 3 (14B)	Qwen 3.5 (27B)	Qwen 2.5 (32B)	GPT-OSS (120B)
EX (%)	70.6	76.7	79.5	70.6
SQL Validity (%)	87.3	95.1	90.5	88.7
First-Attempt Success (%)	41.2	60.8	60.8	68.6
Avg Retries / Query	2.6	1.9	1.5	1.4
Median Latency(s)	8	8.2	11.8	57.3

Table 3. Cross-model comparison (Munich LoD2 dataset)

The latency differences reflect model size and infrastructure: Qwen 2.5 (32B) responds with a median of 11.8 seconds, while GPT-OSS (120B) requires over 57s per query due to its size. Ministral 3 (14B) and Qwen 3.5 (27B) fall in between at approximately 8 and 8.2s respectively. The substantial latency of larger models highlights an important pattern: GPT-OSS-120B achieves slightly better first-attempt success but at a cost that may be to slow for interactive use, whereas Qwen 2.5 delivers the best accuracy with the fastest response time.

6. Discussion

The evaluation demonstrates that automated domain context generation enables LLM agents to query semantic 3D city models with high SQL validity (92–96%) across diverse datasets, languages, and model sizes. Several findings are worth the discussion. First, the cross-model results show no clear correlation between parameter count and execution accuracy: the 14B model matches the 120B model at 70.6% EX, while the 32B model outperforms both at 79.5%. This suggests that for structured database querying, the quality of the assembled domain context is more important than the model capacity. The practical implication is that the framework can be deployed with smaller, local models without significant accuracy loss. This is an important consideration for organizations handling sensitive urban data. In addition to that, SQL validity remains consistently above 92% across five datasets from three countries, three languages, and two levels of detail. The framework handles the transition from LoD2 to LoD3 gracefully. Because despite a 91% increase in object class tokens, total context grows only by 33%, and SQL validity improves slightly (93.5% to 95.7%). This efficiency stems from the three-stage dynamic selection process: object class discovery eliminates non-instantiated classes (reducing hundreds of CityGML-defined classes to only those present), property validation removes schema-defined properties absent from the data, and generic attribute grouping compresses repetitive attributes via prefix aggregation. Together, these steps achieve 98.6–99.8% token reduction compared to unoptimized baselines, keeping the total context under 6,200 tokens across all

This contribution has been peer-reviewed. The double-blind peer-review was conducted on the basis of the full paper.
https://doi.org/10.5194/isprs-annals-XII-4-W1-2026-211-2026 | © Author(s) 2026. CC BY 4.0 License.

tested datasets, which fits within the limits of even small local models with 8k context windows. However, SQL execution accuracy drops notably for Very High complexity queries (47–62%), particularly for the LoD3 dataset. Furthermore, the benchmark queries were designed by domain experts. However, evaluating non-expert user groups queries remains subject to a future user study.

Additionally, to demonstrate that the assembled domain context supports agents beyond natural language querying, we implemented a second MCP client: a data quality agent. This agent connects to the same MCP server, receives the domain context, and evaluates the completeness and consistency of the stored CityGML data. Using the database description provided by the MCP server, this agent autonomously generates SQL queries to evaluate missing property values, incorrect property datatypes, and geometric validity across all feature types, without any manual configuration. The agent produces a structured quality report by using the same domain context that guide the query agent. This demonstrates that the assembled context is not task-specific but serves as a reusable foundation for diverse agent applications over semantic 3D city models.

7. Conclusion and Future work

This paper presented a framework for enabling AI agents to interact with semantic 3D city models through automated domain context generation. The framework takes advantages of the Model Driven Architecture (MDA) and leverages the data structures and semantics of the CityGML 3.0 conceptual model via the 3DCityDB without manual intervention. By separating the 3DCityDB input into static and dynamic components, the framework produces compact, dataset-specific prompts that reduce context size by 98.6–99.8% compared to unoptimized baselines. The domain context is exposed via the Model Context Protocol. Since the MCP server dynamically composes the prompt from whatever classes, properties, and generic attributes are present in the database, the framework inherently supports any CityGML Application Domain Extension (ADE) without modification. The evaluation across five CityGML datasets from three countries demonstrated consistent SQL validity above 92% regardless of language or schema complexity. Moreover, using different LLMs with the same database showed that a 14B parameter model achieves comparable accuracy to a 120B model when given the same assembled context, confirming that prompt quality matters more than model size for structured database querying. A data quality assessment agent reusing the same context further demonstrated the framework's applicability beyond interactive querying.

Future work will address extending the MCP server with database write tools to support attribute updates and geometry modifications, enabling what-if scenario analysis directly through natural language. In addition to that, adapting the framework to other geospatial database schemas, particularly IFC-based building information models, will establish a broader foundation for AI-assisted interaction with the built environment.

References

Chase, H. (2022). *Langchain*. GitHub. https://github.com/langchain-ai/langchain/

Chen, L., Silvennoinen, H., De Wolf, C., Hall, D., Van Mele, T., & Block, P. (2024). *Towards Automated Building Life Cycle Assessments: A Novel Approach Using Large Language Models and the COMPAS Framework*. In Proceedings of IASS Annual Symposia (pp. 1-12). International Association for Shell and Spatial Structures (IASS).

Du, C., Esser, S., Nousias, S., & Borrmann, A. (2026). Text2BIM: Generating Building Models Using a Large Language Model-Based Multiagent Framework. *Journal of Computing in Civil Engineering*, *40*(2), 04025142.

Gröger, G., Kolbe, T. H., Nagel, C., & Haefele, K.-H. (2012). OGC City Geography Markup Language (CityGML) Encoding Standard (pp. 12–019). Open Geospatial Consortium.

ISO. (2024). *ISO 16739-1:2024: Industry Foundation Classes (IFC) for data sharing in the construction and facility management industries — Part 1: Data schema*. ISO. https://www.iso.org/standard/84123.html

Jiang, G., Ma, Z., Zhang, L., & Chen, J. (2024). EPlus-LLM: A large language model-based computing platform for automated building energy modeling. *Applied Energy*, *367*, 123431. https://doi.org/10.1016/j.apenergy.2024.123431

Kanna, K., & Kolbe, T. H. (2025). *Advanced User Interaction with Urban Digital Twins using Large Language Models*. ISPRS Geospatial Week 2025, Photogrammetry & Remote Sensing for a Better Tomorrow.

Kutzner, T., Chaturvedi, K., & Kolbe, T. H. (2020). *CityGML 3.0: New Functions Open Up New Applications. PFG – Journal of Photogrammetry, Remote Sensing and Geoinformation Science, 88(1), 43–61.*

Li, Z., & Ning, H. (2023). Autonomous GIS: The next-generation AI-powered GIS. *International Journal of Digital Earth*, *16*(2), 4668–4686.

Liu, J. (2022). *LlamaIndex*. GitHub. https://github.com/run-llama/llama_index

Liu, S., & Wang, C. (2025). KCitychatBot: A knowledge graph based chatbot system for large-scale CityGML dataset. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XLVIII-4/W15-2025*, 99–105.

Liu, X., Shen, S., Li, B., Ma, P., Jiang, R., Zhang, Y., Fan, J., Li, G., Tang, N., & Luo, Y. (2025). A Survey of Text-to-SQL in the Era of LLMs: Where Are We, and Where Are We Going? *IEEE-Transactions on Knowledge and Data Engineering*, *37*(10), 5735–5754.

Mansourian, A., & Oucheikh, R. (2024). ChatGeoAI: Enabling Geospatial Analysis for Public through Natural Language, with Large Language Models. *ISPRS International Journal of Geo-Information*, *13*(10), 348.

MCP Specification. (2025). Model Context Protocol. https://modelcontextprotocol.io/specification/2025-11-25

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). *Toolformer: Language Models Can Teach Themselves to Use Tools*. Advances in neural information processing systems, 36, 68539-68551.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). *ReAct: Synergizing Reasoning and Acting in Language Models*.

Yao, Z., Nagel, C., Kendir, M., Willenborg, B., & Kolbe, T. H. (2025). The New 3D City Database 5.0—Advancing 3D City Data Management based on CityGML 3.0. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *X-4/W6-2025*, 241–248.

Yao, Z., Nagel, C., Kunde, F., Hudra, G., Willkomm, P., Donaubauer, A., Adolphi, T., & Kolbe, T. H. (2018). 3DCityDB - a 3D geodatabase solution for the management, analysis, and visualization of semantic 3D city models based on CityGML. *Open Geospatial Data, Software and Standards*, *3*(1), 5.

Zhang, Y., Wei, C., He, Z., & Yu, W. (2024). GeoGPT: An assistant for understanding and processing geospatial tasks. *International Journal of Applied Earth Observation and Geoinformation*, *131*, 103976.