

Article

AI Agent Framework Research for Dynamic BIM Query via Retrieval-Augmented Code Generation for IFC

Taewook Kang

Korea Institute of Civil Engineering and Building Technology, Goyang-si 10223, Republic of Korea; ktw@kict.re.kr

Abstract

Accessing and retrieving specific information from complex Building Information Modeling (BIM) data, particularly within the Industry Foundation Classes (IFC) schema, remains a challenge for Architecture, Engineering, and Construction (AEC) professionals. Existing methods, including manual parsing, database conversion, and early-stage AI applications, often suffer from high overhead, inflexibility, or reliance on pre-processed, static data views. This study introduces a novel AI agentic framework that leverages a Large Language Model (LLM) to dynamically generate executable code for BIM data query. The core of this framework is a Retrieval-Augmented Generation (RAG) pipeline that, instead of retrieving factual data, retrieves relevant code exemplars demonstrating the use of IFC parser. The LLM-based agent analyzes a user's natural language query, retrieves functionally relevant code snippets from a specialized vector database, and synthesizes these examples into a tailored, executable script. This "BIM code-retrieval" approach eliminates the need for IFC conversion using databases, allowing the agent to directly use the full power of the IFC parser to obtain the data it needs, and for the LLM to make appropriate inferences to the question. The proposed system demonstrates flexibility and query power compared to methods reliant on static data abstractions. It can handle complex queries that are intractable for previous RAG-based approaches. This AI agent code generation framework demonstrates the cost-effective, inference-based human–BIM interaction method in situations where IFC-to-database conversion is difficult. The prototype of study was developed to analyze the performance of the proposed BIM AI Agent framework and its strengths and weaknesses were examined.

Keywords: Building Information Modeling (BIM); Large Language Models (LLM); Retrieval-Augmented Generation (RAG); code generation; agentic systems; Industry Foundation Classes (IFC)



Academic Editor: Salvatore
Antonio Biancardo

Received: 25 June 2026

Revised: 7 July 2026

Accepted: 8 July 2026

Published: 13 July 2026

Copyright: © 2026 by the author.
Licensee MDPI, Basel, Switzerland.
This article is an open access article
distributed under the terms and
conditions of the [Creative Commons
Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

The Architecture, Engineering, and Construction (AEC) industry has increasingly adopted Building Information Modeling (BIM) as a central data hub, integral to the planning, design, construction, and operational phases of a project's lifecycle. The Industry Foundation Classes (IFC) standard has been pivotal in this transition, providing a vendor-neutral, open-specification data model that ensures interoperability between disparate software platforms [1,2]. But the very same universality and comprehensiveness that has made IFC so popular also poses significant challenges to data retrieval. The IFC schema is an object-oriented, complicated graph-like structure of immense complexity, where information is often deeply nested and linked through intricate relationships. For non-specialist stakeholders, such as project managers or facility operators, directly querying this complex

schema to extract specific, actionable information is infeasible. In addition, IFC continues to expand to domains such as bridges and roads to represent construction life cycle information. This causes the initially designed sub-foundation to be extended and bear more load to support the heavier superstructure, making IFC increasingly difficult for users to handle. The cost, time, and specialized expertise required to parse and analyze IFC files remain a considerable barrier to realizing the full value of BIM.

Over the years, several paradigms have emerged to address this challenge, each with its own set of trade-offs. Traditional methods involve developing custom parsers or implementing an Extract, Transform, Load (ETL) process to move IFC data into more query-friendly formats like relational or NoSQL databases. While this makes data accessible via standard query languages against the database's predefined schema, the process is labor-intensive, requires significant upfront development, and results in a static data snapshot that can quickly become outdated as the source model evolves. A more advanced approach involves transforming the IFC schema into formal ontologies (e.g., OWL) or knowledge graphs, enabling powerful semantic queries using languages like SPARQL [3]. While these methods offer high fidelity, they are often constrained by rigid ontological frameworks and demand specialized expertise in graph technologies, limiting their practical adoption.

With the advent of Large Language Models (LLMs), new possibilities have emerged for natural-language interaction with BIM data [4]. In parallel, retrieval-augmented code generation, which retrieves exemplar code rather than data to condition the model, has shown promise for domain-specific programming tasks [5], motivating its application to IFC querying. An initial approach, explored in a predecessor to this study, utilized Retrieval-Augmented Generation (RAG) to build a BIM knowledge expert agent [1]. This system pre-processed an IFC file by extracting key objects and their associated property sets (PSETs) into simplified textual "chunks." When a user posed a query, the RAG system would retrieve the most relevant textual chunks and feed them to an LLM to synthesize an answer. While this marked an improvement in usability, its capability was fundamentally constrained by the scope of the pre-processed data. It could only answer questions about information that had been explicitly extracted during the "BIM chunking" phase, failing to address queries that required navigating relationships or accessing data outside the pre-selected PSET [1].

These existing solutions share a common philosophical approach: they attempt to simplify the data for the LLM. They operate by creating an abstraction layer, whether a database schema, a knowledge graph, or a set of text chunks, between the LLM and the raw, complex IFC data.

This research proposes a new method that moves from data retrieval to skill code retrieval. The core idea is to augment the LLM not with data from the BIM model, but with procedural knowledge, in the form of code examples, of how to acquire those facts using specialized tool code. This approach removes the static data abstraction layer and instead provides the LLM with direct, mediated access to the raw IFC file through the parser's code. The system's intelligence is no longer limited by a pre-defined and lossy data transformation, but by the functional completeness of the IFC parser and the LLM's ability to learn and compose its functions.

This study makes the following key contributions to the field.

1. It presents the design, implementation, and analysis of a novel agentic method, detailed in the code-based BIM AI agent framework, which generates executable code from natural language queries to dynamically interrogate IFC files directly.
2. It provides a comparative analysis demonstrating the proposed method's flexibility and expanded query capabilities over data-centric RAG approaches that rely on pre-processed, static data views.

3. It offers a critical discussion of the practical implications of this paradigm shift, including a rigorous examination of the trade-offs between per-query computational costs and preprocessing overhead, and a crucial analysis of the security imperatives associated with executing LLM-generated code.

2. Study Methodology

This study adopts a systematic methodology for developing and evaluating an AI agent framework for dynamic inquiry and inference of BIM from IFC, as depicted in the accompanying Figure 1. This process comprises the following key stages.

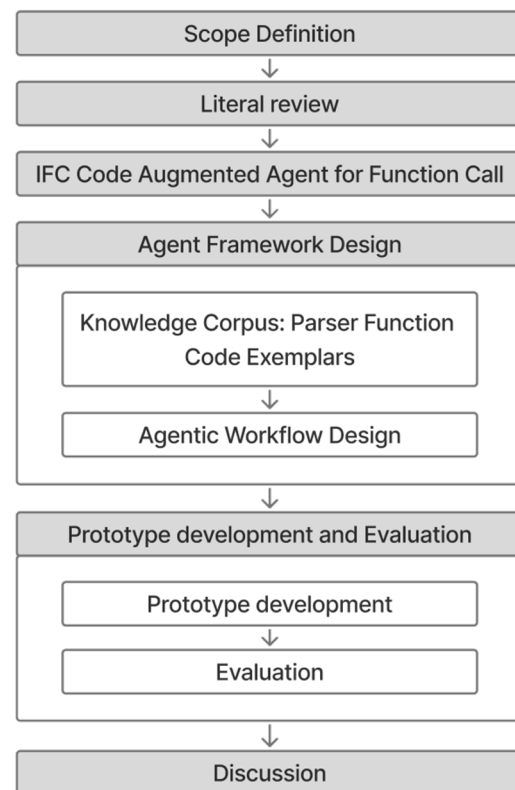


Figure 1. Research process.

First, in the Scope Definition stage, the research objectives and scope are established. This involves identifying the limitations of existing methods for accessing and retrieving complex information within BIM data, particularly the IFC schema, and defining the necessity for a novel AI agent framework to address these challenges.

Next, a Literature Review is conducted to ascertain the state-of-the-art in relevant research domains in natural language interfaces for BIM, retrieval-augmented code generation, and LLM-powered agentic systems. This analysis contributes to defining the unique contributions of the present study.

Subsequently, the core concept of the IFC Code Augmented Agent for Function Call is established. In this stage, the idea of augmenting the LLM with procedural knowledge, through the use of code from a specialized tool for direct data acquisition from BIM models, is elaborated upon, rather than relying on direct data retrieval.

Following this, the Agent Framework Design stage commences, which is divided into two primary components. First, a Knowledge Corpus: Parser Function Code Exemplars is constructed. This corpus serves as a skill code library for interacting with building models, rather than a collection of factual building data. Second, the Agentic Workflow Design is

developed, outlining the specific operational flow of the agent to transform a user's request into a verified answer.

The subsequent stage is Prototype development and Evaluation. This involves Prototype development to implement the designed framework. Following development, an Evaluation is conducted to analyze the effectiveness of the developed system. This includes the case-study validation analysis to demonstrate the system's ability to handle queries of varying complexity, and a structured methodological comparison to situate the proposed approach against key alternatives, highlighting its unique advantages and trade-offs.

Finally, in the Discussion stage, a comprehensive analysis of the research findings is provided. This explores the benefits of dynamic API interaction, critical security and reliability considerations associated with executing generated code, and the associated performance. Furthermore, this section identifies the limitations of the current framework and suggests avenues for future research.

3. Related Work

AI and LLM are driving automation across various industries based on their vast knowledge reasoning capabilities, but they still face limitations in interpreting closed data in specific domains (e.g., complex 3D architectural objects like BIM) due to hallucination and a lack of expertise [6]. To address this, existing literature has primarily introduced RAG frameworks or autonomous agents that build external databases to inject static knowledge [7]. In the BIM field, there has been research applying this approach to develop BIM-based AI agent technology. This study proposes a method that utilizes BIM files directly rather than searching through such RAG-based databases. This research classifies existing research into three key trends and analyzes the differences and developmental relationships between each trend and the framework presented in this new paper.

3.1. Intelligent Automation in the AEC Field

A key challenge for the existing AEC industry has been providing natural language interfaces (NLI) for vast BIM and IFC data that are difficult to handle without expertise. To this end, virtual assistants that automatically convert users' natural language queries into structured database queries have been actively researched [4,8]. In addition, technologies have been developed to map complex attributes and relational constraints to SPARQL queries using ontology-based semantic parsers [9], or to infer topological relationships of building objects by applying Graph-RAG after converting IFC data into a graph structure [3,10]. Workflow research involving the introduction of multi-agents to analyze IFC data and assess risks has also yielded results [11]. LLM-based automation of commercial construction scheduling has also been explored [12]. Furthermore, research has focused on technology acceptance analysis using the Unified Theory of Acceptance and Use of Technology (UTAUT) extension model [13] and expert-level knowledge response systems utilizing RAG [1].

All existing BIM NLI and Graph-RAG studies had a limitation in that they required a heavy preprocessing step of converting complex IFC files into relational databases, ontology, or graph structures (Parsing/Mapping) beforehand.

In contrast, this study aims to eliminate the pre-DB conversion step that causes this overhead. The framework of the new paper searches for 'code exemplars' that demonstrate how to use the IFC parser, instead of directly searching for IFC data in the vector DB. Subsequently, an LLM-based agent synthesizes and executes a customized script in real-time that utilizes the IFC parser in response to the user query, thereby directly extracting only the desired information.

3.2. Trends in Multi-Agents Based on Software Engineering (SE)

In the field of software engineering, LLM supports the software development life-cycle, including code generation, translation, and review. Existing studies have maximized code generation performance by applying RAG frameworks and demonstrated cost-effectiveness through the Sequential Integration Fusion (SIF) strategy [5]. In particular, research stands out for reducing contextual errors by combining few-shot learning and RAG when translating legacy code (e.g., Fortran) into C++ [14], and for overcoming Low-Frequency Ground-Truth Tokens (LFGT), which are difficult to generate probabilistically during code review, using RAG [15]. In addition, evolving knowledge base systems like the Evolving Retrieval (EVOR) pipeline [16] and LLM multi-agent (LMA) systems, which divide labor from planning to testing, are minimizing human developer intervention [17]. The need to establish benchmarks specialized for Low-Resource Programming Languages (LRPLs) has also been a major topic of discussion [18,19].

In existing trends, Retrieval-Augmented Code Generation (RACG) technology has focused strictly on the fundamental purposes of software engineering, such as software development, translation, and maintenance.

This study applied this as a BIM interpretation code search and generation mechanism to solve the question-answering problem in the BIM domain. Here, we investigate agents that go beyond LLMs simply responding with text or writing general-purpose application code, and instead actively program and manipulate specialized domain libraries (IFC Parser).

3.3. Autonomous Agent Fundamentals, Cognitive Models, and Security Reliability

Large Language Model-based agent research has demonstrated how LLMs achieve autonomy by systematizing systems into profiling, memory, planning, and behavior modules [7]. However, as agents autonomously utilize external plugins and tools, critical security vulnerabilities such as indirect prompt injection and data leakage have emerged [20]. To defend against this, intelligent defense agents are being designed that combine the AutoGen framework with RAG to identify OWASP vulnerabilities and enforce an organization's security policies in real time [21], and improving reliability in distributed computing environments is also being addressed as a major challenge [22].

This study differs in that it focuses on practical implementations that demonstrate the effectiveness and flexibility of complex query processing, rather than focusing on existing cognitive [23], philosophical, or defensive security research. However, from a trend perspective, the proposed "structure in which LLM dynamically generates BIM analysis code and directly executes it in the system environment to obtain results" is highly likely to face OWASP security vulnerabilities, which are strongly warned against in existing security literature [20]. Therefore, for the framework of this study to be safely adopted in actual industries, it is expected that techniques to verify code generated by LLM in a secure isolated environment (Sandbox) and prevent malicious prompt injection will become increasingly important.

Existing research trends have evolved along the axes of improving BIM accessibility through NLI and Graph-RAG, automating software development through RACG and multi-agents, and strengthening the security of autonomous agents. The proposed method presents a mechanism that bypasses the BIM database preprocessing costs of these existing studies. Unlike existing research, this method proposes a mechanism to search for and dynamically execute analysis code that handles BIM data.

Therefore, this study stands at the intersection of these research fields and demonstrates how to solve problems arising in BIM data retrieval in AEC research by using agent architecture from AI research and RAG-for-code mechanism from NLP research.

4. RAG-Augmented Agent for Code Function Call of IFC Parser

The proposed framework is designed to function as an intelligent intermediary between a non-technical user and a complex IFC data file. It translates natural language queries into executable code that leverages IFC parser to perform the requested data extraction.

4.1. Agent Framework Design

The system is architected as a modular pipeline orchestrated by LLM, which facilitates the interaction between the LLM, the retrieval system, and the execution environment. The agent framework for BIM query from natural language interface is depicted in Figure 2.

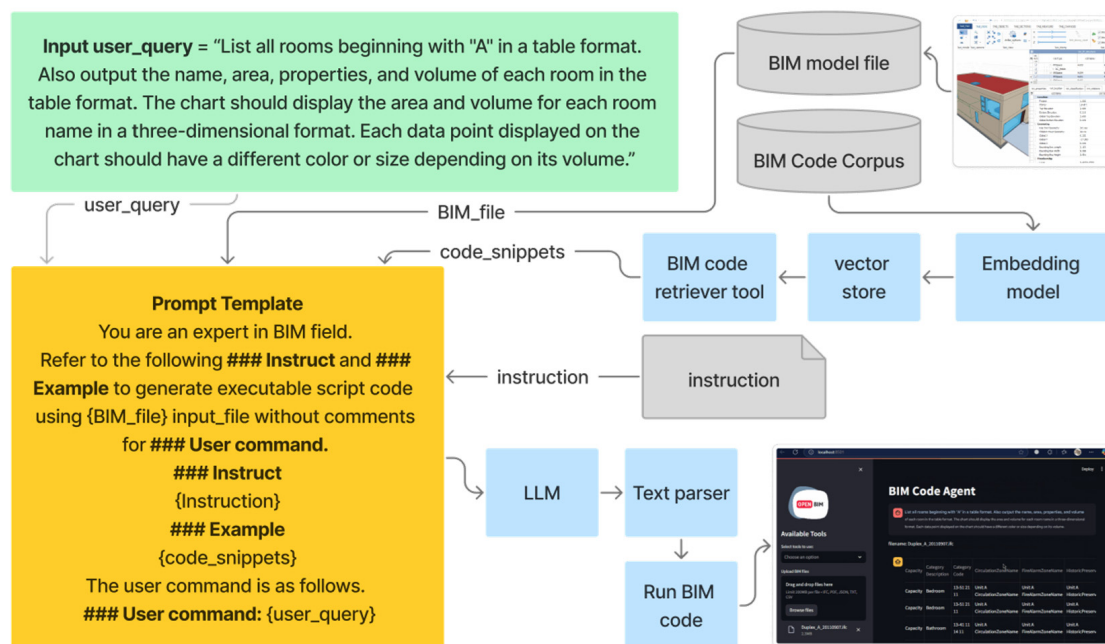


Figure 2. BIM Code Agent Framework for BIM Query.

The framework is architected as a modular pipeline, which facilitates the interaction between the LLM, the retrieval system, and the execution environment. The overall workflow is depicted in the accompanying diagram.

The process begins when a user submits a natural language query (e.g., “List all doors on the ground floor”) and uploads an IFC file. This input, along with the BIM model file and code snippets files, feeds into the system. The user’s query is directed to the BIM code retriever tool. This tool, utilizing an Embedding model, performs a similarity search on a specialized vector store containing embeddings of code examples.

The top-ranked, most relevant code snippets are retrieved. These retrieved snippets, along with an “instruction” (likely derived from the user’s query or a predefined system instruction), are then synthesized into a Prompt Template. This augmented prompt, rich with contextual examples, is then sent to the LLM. The LLM processes this comprehensive prompt and is instructed to generate a complete, executable BIM code script to answer the query.

The generated script is not executed directly within the LLM. Instead, it is passed to a secure, isolated sandboxing environment, represented by the “Run BIM query code” component. This sandbox executes the code and captures its standard output, which contains the answer to the user’s query. This output is then relayed back to the user, likely through a “Text parser” and ultimately presented via a user interface, such as the “BIM Code Agent” web application shown in the diagram.

This architecture represents a fundamental departure from earlier data-centric RAG pipelines. It replaces intensive preprocessing steps, such as parsing the entire BIM file like IFC, performing “BIM chunking” to extract textual facts, and embedding these facts into a vector database, with a lighter-weight process focused on retrieving procedural knowledge (code). This shifts the computational load from an intensive preprocessing stage to the query-time execution stage, enabling dynamic interaction with the live IFC file.

4.2. Knowledge Corpus: Parser Function Code Exemplars

The efficacy of the Retrieval-Augmented Generation (RAG) mechanism is entirely dependent on the quality and structure of its knowledge corpus. In this framework, the corpus is not a collection of facts about a specific building but rather a library of skills for interacting with any building model. This corpus consists of a curated collection of parser function code snippets stored in a dedicated code sample directory for RAG. The proposed methodology is designed to be independent of any specific underlying parser library, allowing for adaptability.

A core design principle governing this corpus is atomicity and composability. Each code snippet is intentionally small and self-contained, engineered to effectively solve one discrete task. For example, one snippet might illustrate how to iterate through elements of a certain type, another how to locate the spatial container (e.g., the storey) of an element, and a third how to read properties from an element’s property sets using a parser library like ifcopenshell. This atomic design empowers the LLM to function as a composer. For a complex query, such as “Find the fire rating of all doors on the ground floor,” the LLM can retrieve and combine functionalities from multiple atomic snippets: one for identifying the correct floor, one for recognizing doors on that specific floor, and another for extracting the fire rating property from each identified door. Table 1 provides illustrative examples of these atomic code exemplars.

Table 1. Structure of the BIM Code Knowledge Corpus (IFC exemplars).

Snippet ID	Description	Key Function(s)	Function Usage Example
CS-01	Load IFC file	ifcopenshell.open(“input_BIM_file”)	ifc_file = ifcopenshell.open(‘input_BIM_file ’) settings = ifcopenshell.geom.settings()
CS-02	Get all elements of a specific IFC type.	model.by_type(“IfcType”)	for wall in model.by_type(“IfcWall”): ...
CS-03	Get all properties from an element’s PSETs.	pset = relationship.RelatingPropertyDefinition	for relationship in product.IsDefinedBy: ...
CS-04	Find the spatial geometry of an element.	body = ifcopenshell.util.representation.get_representation(element, “Model”, “Body”)	element = ifc_file.by_type(‘IfcWall’)[0] ...

To characterize the corpus systematically, the table below reports its functional categories together with the IFC entities and ifcopenshell APIs each category exercises. The corpus is curated under three principles: atomicity (one discrete task per snippet), composability (snippets combine for multi-hop queries) and parser-agnosticism.

4.3. Agentic Workflow Design

The agent executes a four-step workflow to transform a user’s request into a verified answer.

Step 1: Retrieval. Upon receiving a query, the system first embeds it into a high-dimensional vector. This query vector is then used to perform a similarity search against the pre-computed vectors of the code snippets in the knowledge corpus. As supported by the literature on code retrieval, a lexical-focused search algorithm like MMR (Maximal Marginal Relevance) (1) is highly effective for this task, as it excels at matching the specific keywords

corresponding to function names (e.g., `get_BIM_object`), class names (e.g., `IfcDoor`), and parameters that are crucial for correct code generation.

$$MMR(D_i) = \lambda \cdot Sim_1(D_i, Q) - (1 - \lambda) \cdot \max_{D_j \in S} Sim_2(D_i, D_j) \quad (1)$$

Here, Sim_1 measures the relevance of a document to the query, and Sim_2 measures the similarity between documents. The parameter λ ($0 \leq \lambda \leq 1$) controls the tradeoff between prioritizing relevance and prioritizing diversity. The algorithm iteratively selects the highest scoring document and adds it to the result set.

Step 2: Prompt Synthesis. The top-k most relevant code snippets retrieved from the vector store are then dynamically inserted into a master prompt template. This template provides the LLM with its core instructions: to act as an expert developer specializing in BIM, to carefully study the provided code examples to understand the domain-specific parsing library's API, and to synthesize this knowledge to write a single, complete, and executable script that directly answers the user's original query. This technique is a dynamic form of few-shot prompting, where the illustrative examples ("shots") are not fixed but are tailored in real-time to the specific needs of the query.

Step 3: Code Generation. The LLM processes the comprehensive, context-rich prompt. It analyzes the user's intent, identifies the relevant patterns and functions from the retrieved examples, and generates the final Python script. A typical script will include the necessary imports, a line to load the user-provided IFC file, logic for iterating through and filtering entities, calls to the appropriate functions of the domain-specific parsing library, and the final output results like table, graph.

Step 4: Secure Execution. The generated code is inherently untrusted. Executing it in the main application environment would pose a severe security risk. Therefore, the script is passed to a secure sandboxing module for execution. This is a critical step that isolates the code from the host system's file system, network, and environment variables, mitigating a range of potential vulnerabilities, including malicious file operations or data exfiltration. The sandbox executes the script, captures any output sent to stdout and stderr, and returns only these streams to the main application, ensuring that the untrusted code cannot affect the host system.

Step 5: Execution Feedback (Self-Debugging). LLM-generated domain code often fails on first execution due to runtime errors or API misuse. The workflow therefore closes an execution-feedback loop (Figure 3): when the sandbox reports an exception, the captured traceback, the failing script and the original query are returned to the LLM with an instruction to diagnose and correct the code. This repeats up to $R = 3$ iterations (bounded by an 80s wall-clock budget) or until execution succeeds. An ablation shows the loop raises the overall Execution Success Rate from 50.0% to 62.5%, with the largest gains on the relational and complex tiers.

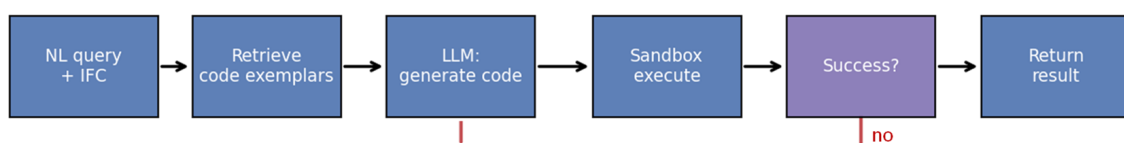


Figure 3. Agent workflow with the execution-feedback (self-debugging) loop.

4.4. Agent Architecture Design

The architecture of the BIM Code Agent is designed as a modular and extensible system, leveraging LLM at its core to facilitate dynamic interaction with BIM data. This design emphasizes the separation of concerns and the composability of various components, enabling robust retrieval-augmented code generation for BIM queries. The system orches-

trates multiple tools and data sources to process natural language queries and generate executable code.

The BIM_code_agent acts as the primary orchestrator, initializing the system, managing the vector store, and handling the overall query processing flow. It delegates specific tasks to a RAG_tool_chain, which embodies the multi-step Retrieval-Augmented Generation process. Central to this RAG process is the vector_store, responsible for managing the embeddings of code snippets that form the BIM_code_corpus (Table 2). The LLM component serves as the reasoning engine, generating code based on structured prompts.

Table 2. Code-exemplar corpus: functional categories and coverage.

Category	Source File(s)	Chunks	#IFC Entities	IFC Entities (Sample)	Ifcopenshell APIs
File/model creation	ifcopenshell_code_create.txt	2	5	IfcBuilding, IfcBuildingStorey, IfcProject, IfcSite, IfcWall	aggregate.assign, api.root/create
Loading, type filtering, property/PSET access	ifcopenshell_code_pset.txt	2	19	IfcBeam, IfcBuildingElementPart, IfcBuildingElementProxy, IfcBuildingStorey, IfcColumn, IfcCovering, IfcDoor, IfcMember, IfcPropertySet, IfcPropertySingleValue, IfcRailing, IfcRelDefinesByProperties	HasProperties, IsDefinedBy, RelatingProperty-Definition, ifcopenshell.open
Geometry and spatial extents	ifcopenshell_code_geometry.txt	1	0	-	by_id, geom.iterator, geom.settings, ifcopenshell.open
File writing/export	ifcopenshell_code_write.txt	0	0	-	-

The system employs a set of specialized RAG_tool implementations, each addressing a distinct functionality within the workflow. These tools include a BIM_code_retriever_tool for code snippet retrieval, an LLM_tool for interfacing with the LLM, a Text_parser_tool for processing text outputs, and a Run_BIM_code_tool for secure code execution. The Prompt_template defines the structure and instructions for guiding the LLM's code generation. The BM_model_file represents the input BIM data that the generated code will interact with.

5. Prototype Development and Evaluation

This section develops the prototype and evaluates the effectiveness of the BIM Code Agent Framework.

5.1. Prototype Design

Figure 4 and Table 3 summarizes the key classes, their roles, and their provided member sets within this architectural design. The prototype of the BIM Code Agent components are designed as a modular system, with its architecture depicted in the package diagram (Figure 5). This design leverages established open-source libraries and components to realize the functionality of the proposed agentic system. In this study, LLM used an openweight model such as qwen2.5-coder:7b.

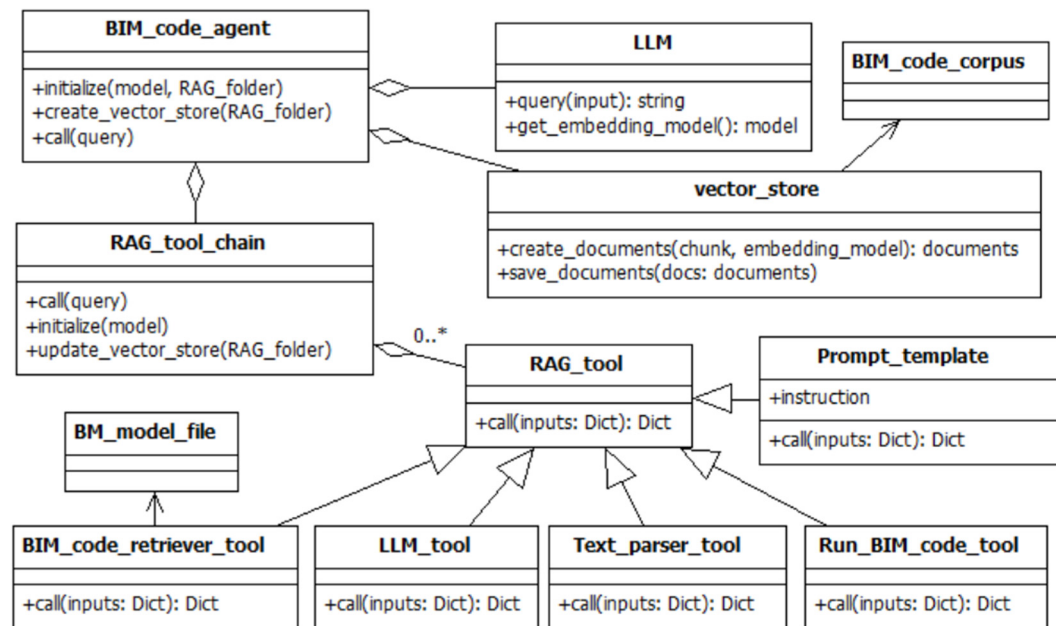


Figure 4. BIM Code Agent Framework architecture (Dict = dictionary datastruct) (UML; * denotes UML multiplicity).

Table 3. Classes summary of BIM Code Agent Framework architecture.

Class Name	Role	Provided Member Set
BIM_code_agent	Main orchestrator; initializes and manages the overall system flow.	initialize(model, RAG_folder) create_vector_store(RAG_folder) call(query)
LLM	Core reasoning engine; generates code and provides embedding models.	query(input): string get_embedding_model(): model
BIM_code_corpus	Repository of code examples and knowledge for BIM operations.	code corpus files
vector_store	Manages and retrieves embedded documents (code snippets).	create_documents(chunk, embedding_model): documents save_documents(docs: documents)
RAG_tool_chain	Orchestrates the sequence of RAG tools for a complete query workflow.	call(query) initialize(model) update_vector_store(RAG_folder)
BM_model_file	Represents the input BIM model data.	BIM model file
RAG_tool	Abstract base class for all specialized tools in the RAG process.	call(inputs: Dict): Dict
BIM_code_retriever_tool	Retrieves relevant BIM code snippets from the vector store.	call(inputs: Dict): Dict
LLM_tool	Interfaces with the Large Language Model for code generation.	call(inputs: Dict): Dict
Text_parser_tool	Processes and formats text outputs.	call(inputs: Dict): Dict
Run_BIM_code_tool	Executes the generated BIM code in a secure environment.	call(inputs: Dict): Dict
Prompt_template	Defines the structure and instructions for LLM prompts.	instruction: string call(inputs: Dict): Dict

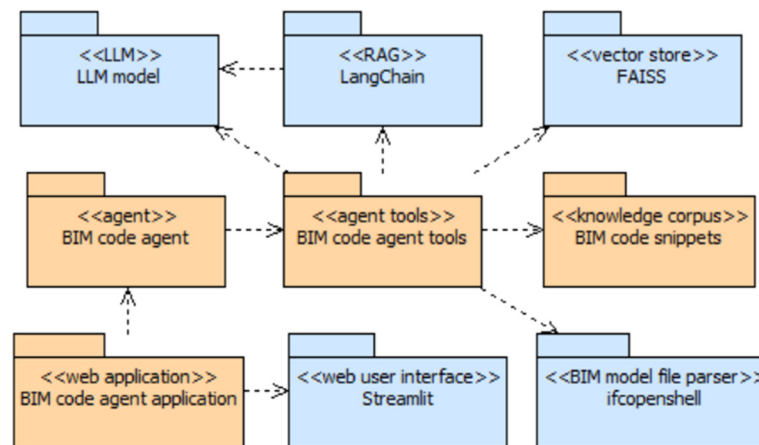


Figure 5. BIM Code Agent package diagram (UML).

At its core, the prototype integrates an LLM (represented as <<LLM>>) which serves as the central intelligence for understanding natural language queries and generating code. This LLM interacts with the <<RAG>> LangChain, which is utilized for orchestrating the Retrieval-Augmented Generation process.

The <<agent>> BIM Code Agent package encapsulates the primary logic and control flow of the agent. This agent interacts with a set of <<agent tools>> BIM code agent tools, which represent the specific functionalities the agent can invoke. These tools, in turn, interact with various other components:

<<knowledge corpus>> BIM code snippets package represents the collection of code examples used for retrieval, forming the basis of the agent’s “skill” acquisition.

<<vector store>> FAISS (Facebook AI Similarity Search) package denotes the specific technology chosen for the vector store, responsible for efficiently storing and retrieving the embeddings of the BIM code snippets.

<<BIM model file parser>> ifcopenshell package represents the specific open-source library used for parsing and interacting with IFC BIM model files. This component provides the BIM data parsing function of the BIM Code Agent-generated code.

For user interaction, the <<web application>> BIM Code Agent application provides the overall application layer. It utilizes <<web user interface>> Streamlit for building the interactive web interface, allowing non-expert users to submit queries and receive results seamlessly. The web application interacts with the BIM Code Agent to process user requests.

The overall prototype was designed and implemented to be modularized for the purpose of stereotyping and to be replaceable with better libraries such as RAG, vector store, BIM parsers.

5.2. Evaluation of Query Capabilities

The framework’s performance lies in its flexibility to address natural language-based user queries without a preprocessing step like database conversion, by composing atomic API calls. This is demonstrated through a series of examples, ranging from simple to complex, which highlight the agent’s ability to interpret user intent and generate corresponding executable code. For testing, the IFC sample file (2.3 MB) modeling a “Duplex Apartment” type building with two floors was used (Figure 6). For reference, this model was selected as a test example because it is useful for executing and reviewing the queries of this study, as it includes 10 spaces with structural elements on the first and second floors, as well as various elements such as floors and ceilings.

- (4) Generate code to format this data into a table.
- (5) Include additional code to create a 3D chart using a visualization library.
- (6) The code for the chart must correctly map the 'area' and 'volume' data points, and apply dynamic styling (color/size) based on the 'volume' value.

List the rooms starting with the 'A' name in a table format. The name and area attributes of each room are also printed out in table format.

name	type	Reference	CeilingCovering	WallCovering	FloorCovering	Number	Name	OmniClass Table 13 Category	Level	Upper Limit	Limit Offset	Base Offset	Area
A101	IfcSpace		CeilingCovering	WallCovering	FloorCovering	A101	Hearth	13-51 24 11: General Residential Space	Level 1	Level 1	2.6	0.0	17.936237
A201	IfcSpace		CeilingCovering	WallCovering	FloorCovering	A201	Hallway	13-51 24 11: General Residential Space	Level 2	Roof	0.0	0.0	7.799955

Figure 8. Relational Query's result in BIM.

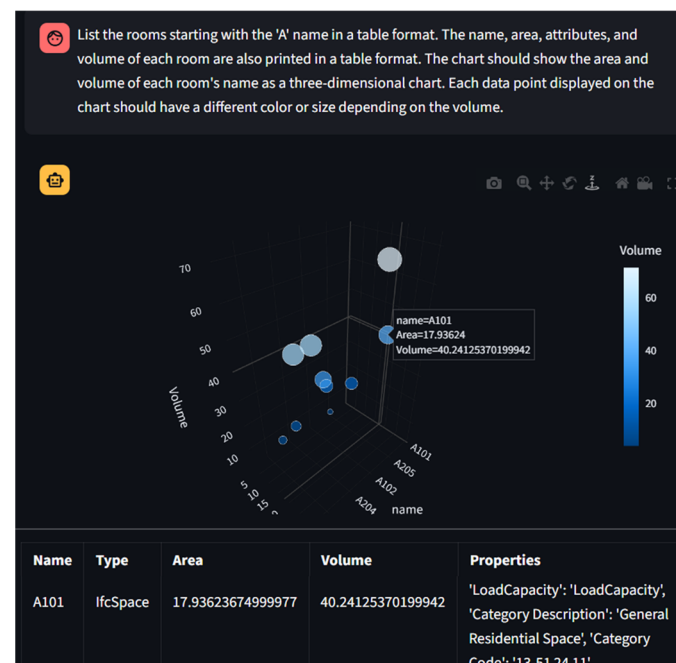


Figure 9. Complex query's result in BIM.

Table 4 reports the ablation of the execution-feedback (self-debugging) loop introduced in Section 4.3, contrasting Execution Success Rate (ESR) and Answer Accuracy with and without the loop, together with the average number of self-debugging iterations, across the three query tiers. Table 5 provides a transparent, step-by-step trace of how the agent processes such a complex query, illustrating the direct link between the retrieved knowledge and the final generated code.

Table 4. Ablation of the execution-feedback loop with ESR, accuracy.

Tier	ESR (No Loop)	ESR (Loop)	Acc. (No Loop)	Acc. (Loop)	Avg. Iters
Simple	100.0%	100.0%	100.0%	100.0%	1.00
Relational	37.5%	62.5%	37.5%	37.5%	2.38
Complex	12.5%	25.0%	0.0%	17.5%	3.38
Overall	50.0%	62.5%	45.8%	51.7%	2.25

Table 5. BIM code retrieval and generation steps for complex queries.

Component	Content
User Query Prompt	“List the rooms starting with the ‘A’ name in a table format. The name, area, attributes, and volume of each room are also printed in a table format. The chart should show the area and volume of each room’s name as a three-dimensional chart. Each data point displayed on the chart should have a different color or size depending on the volume.”
Retrieved Snippet	<pre>def get_object_as_name(ifc_file, name_prefix): return [obj for obj in ifc_file.by_type('IfcSpace') if obj.Name.startswith(name_prefix)] def get_properties(ifc_file, element): properties = {} for definition in element.IsDefinedBy: if definition.is_a('IfcRelDefinesByProperties'): property_set = definition.RelatingPropertyDefinition if property_set.is_a('IfcPropertySet'): for prop in property_set.HasProperties: properties[prop.Name] = prop.NominalValue.wrappedValue return properties</pre>
Synthesized Code 1 (BIM input file load)	<pre>ifc_file = ifcopenshell.open('./expert_kb_files/duplex_a_20110907.ifc') spaces = get_object_as_name(ifc_file, 'A')</pre>
Synthesized Code 2 (Property Query)	<pre>process_list = [] for space in spaces: obj = { 'name': space.Name, 'type': space.is_a(), 'Area': getattr(space, 'GrossFloorArea', None), 'Volume': getattr(space, 'Volume', None) } obj.update(get_properties(ifc_file, space)) process_list.append(obj)</pre>
Synthesized Code 3 (Plot Graph)	<pre>result_fig = px.scatter_3d(result_df, x='name', y='Area', z='Volume', color='Volume', size='Volume')</pre>
Final Output (Plot Table)	<pre>result_markup = '<table>' result_markup += '<tr><th>Name</th><th>Type</th><th>Area</th><th>Volume</th><th>Properties</th></tr>' for obj in process_list: result_markup += f'<tr><td>{obj["name"]}</td><td>{obj["type"]}</td><td>{obj.get("Area", "N/A")}</td><td>{obj.get("Volume", "N/A")}</td><td>{obj}</td></tr>' result_markup += '</table>'</pre>

Figure 10 illustrates the prototype’s successful execution of a complex, multi-faceted query, demonstrating the framework’s ability to handle intricate user requests.

The user’s natural language query, “List the rooms whose names begin with ‘A’ in a table format. Print the name, area, properties, and volume of each room in a table format. The chart should display the area and volume for each room name in a 3D chart. Each data point displayed on the chart should have a different color or size depending on its volume,” is processed by the BIM Code Agent.

The prototype’s web-based user interface, developed with Streamlit, allows the user to upload a BIM file (e.g., “Duplex_A_20110907.ifc”). Upon submission of the query, the agent performs a Retrieval-Augmented Generation (RAG) process, synthesizing a Python script that addresses all parts of the user’s request.

The generated script successfully navigates the IFC file to identify rooms beginning with ‘A’. It then extracts specific data, including PSet_Revit_Dimensions and PSet_Revit_Identity, for each room (e.g., room ‘A103’).

The results are presented in a two-part output.

Tabular Results: A table lists the requested information for each identified room, including name, area, and volume, confirming the agent’s ability to perform accurate attribute lookups and data aggregation.

3D Chart Visualization: The prototype generates a three-dimensional chart that visually represents the area and volume of the rooms. Crucially, the data points on this chart are dynamically colored and sized according to the room’s volume, as specified in the query. This demonstrates the agent’s advanced capability to not only retrieve and process data but also to generate code for sophisticated, multi-dimensional data visualization and analysis.

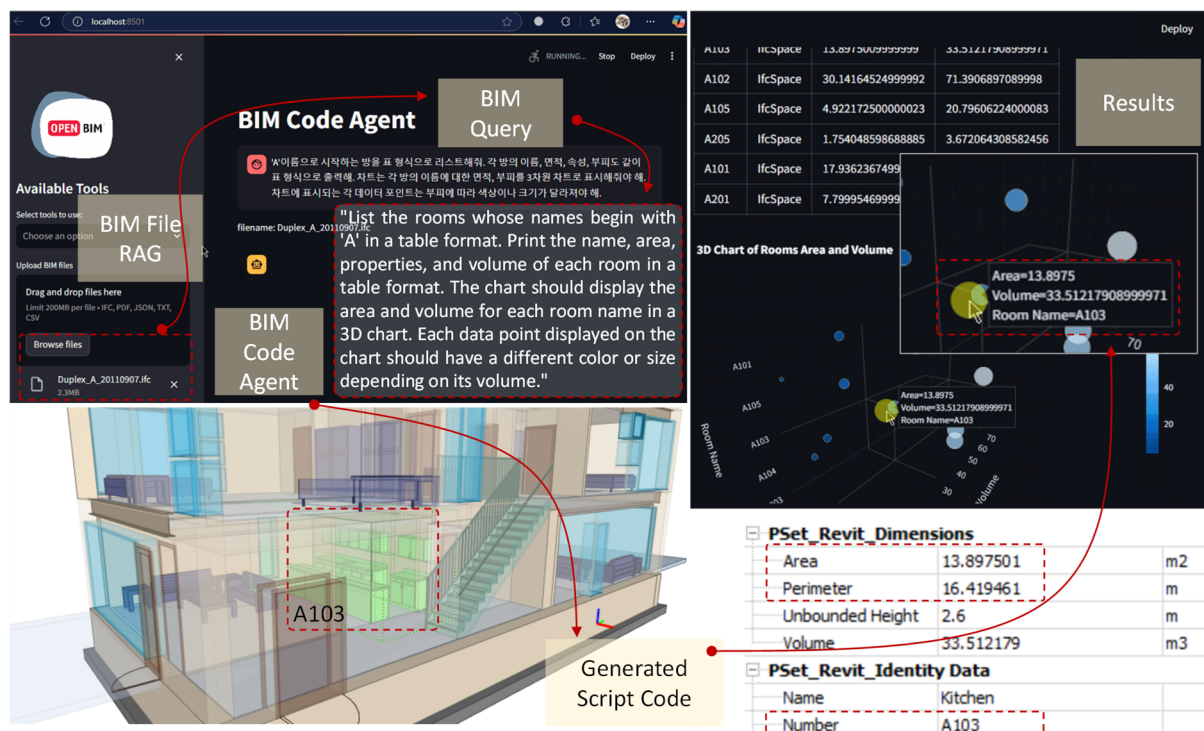


Figure 10. BIM query and results in BIM Code Agent (complex query; the dashed box highlights the query result area, and the Korean text is incidental user-interface language).

The successful execution of this complex query, which requires multi-hop reasoning, attribute retrieval, and external library calls for data visualization, validates the effectiveness and flexibility of the proposed agentic framework.

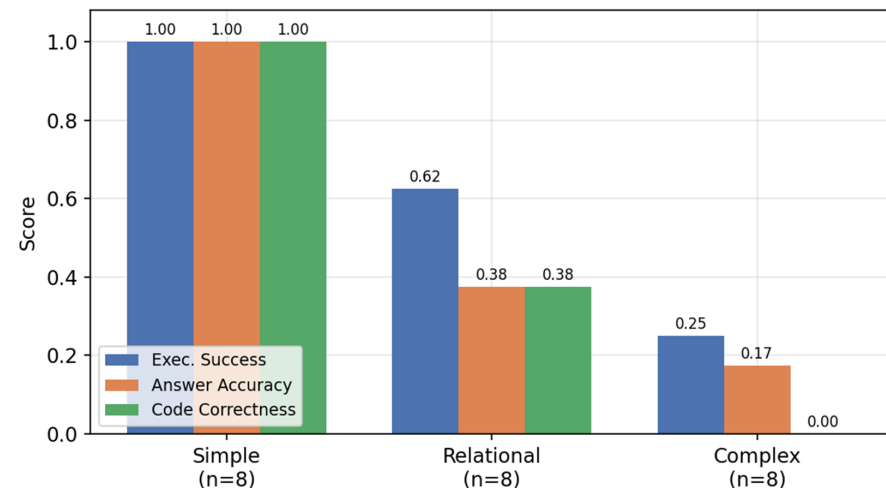
5.3. Performance Evaluation

To complement the qualitative walk-throughs of Section 5.2 with objective metrics, we constructed a standardized benchmark of 24 natural-language queries in three difficulty tiers (eight simple, eight relational, eight complex). Each query is paired with a reference answer computed directly from the IFC model (Appendix A), so the ground truth is measured. The Execution Success Rate, Answer Accuracy, Code Correctness and end-to-end latency were reported. On the proposed method these are 62.5%, 51.7%, 45.8%, 100.0% and 89.12 s overall (Table 6, Figure 11).

The complex tier yields 0.0% Code Correctness despite a 25.0% Execution Success Rate, indicating that the generated programs run without error but implement incorrect logic. This gap arises because complex queries chain multiple IFC navigation and aggregation steps, where a single erroneous intermediate operation silently propagates through the program. The elevated latency of this tier (198.39 s) further reflects unsuccessful iterative repair attempts, marking deeply compositional BIM tasks as the current limit of single-agent code synthesis.

Table 6. Quantitative performance of the proposed method by query tier.

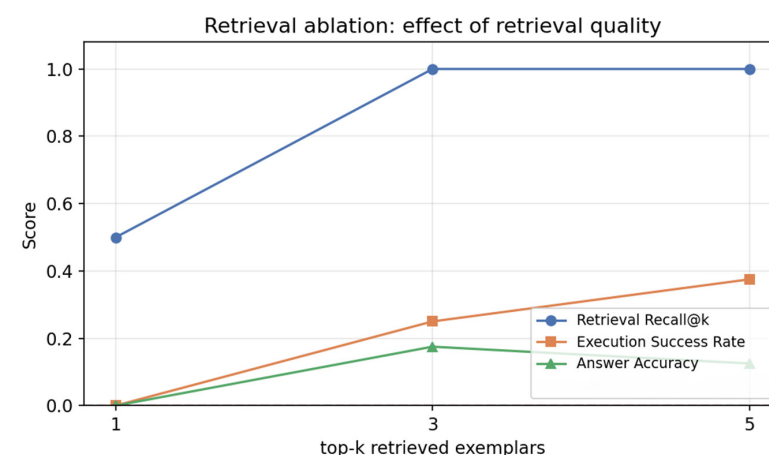
Tier	N	Exec. Success	Answer Acc.	Code Correct.	Latency (s)
Simple	8	100.0%	100.0%	100.0%	17.64
Relational	8	62.5%	37.5%	37.5%	51.32
Complex	8	25.0%	17.5%	0.0%	198.39
Overall	24	62.5%	51.7%	45.8%	89.12

**Figure 11.** Proposed-method metrics by query tier.

Varying the number of retrieved exemplars (top-k) and disabling retrieval isolates the contribution of the corpus. With retrieval off, Execution Success and Answer Accuracy both collapse to 0.0%, confirming that the exemplars, not the base model alone, drive correct IFC navigation; both rise with Recall@k (Table 7, Figure 12).

Table 7. Retrieval ablation (complex tier): effect of top-k and RAG on/off.

top-k	Recall@k	Exec. Success	Answer Acc.
1	50.0%	0.0%	0.0%
3	100.0%	25.0%	17.5%
5	100.0%	37.5%	12.5%
off	0.0%	0.0%	0.0%

**Figure 12.** Effect of retrieval quality on downstream performance.

To comprehensively evaluate the contribution of the proposed agentic code generation approach, it is essential to compare its performance against alternative BIM query methodologies, particularly concerning the quality and executability of generated code for BIM-specific tasks. While general-purpose LLM agents, such as those powering commercial chatbots, possess broad code generation capabilities, they frequently encounter significant challenges when confronted with domain-specific tasks like BIM data querying.

A. Challenges with General-Purpose LLM Agents

When a generic LLM agent (e.g., ChatGPT without specialized context) is prompted to generate code for complex BIM queries (e.g., “List all rooms beginning with ‘A’ and display their area, properties, and volume in a table format, with a 3D chart showing area and volume for each room, colored and sized by volume”), it often produces code that is syntactically flawed or, more critically, semantically incorrect for the BIM domain. Such code typically fails to execute properly, leading to runtime errors, or generates inaccurate results due to a lack of deep understanding of IFC schema navigation and the specific API calls required by BIM parsing libraries. This deficiency stems from their training data’s general nature, which lacks the granular, specialized knowledge of BIM data structures and relevant programming paradigms.

B. Performance of the Proposed BIM Code Agent

In stark contrast, the proposed BIM Code Agent Framework, by leveraging RAG with a curated corpus of domain-specific code exemplars, demonstrates good performance in generating executable and functionally correct code for BIM queries. When presented with the same complex queries, our agent dynamically retrieves the most relevant code snippets, integrates them into a context-rich prompt, and instructs the LLM to synthesize a tailored code script. This process consistently yields code that successfully loads the BIM model, navigates its complex relationships, extracts the requested information, and presents it in the desired format, including complex data visualization requirements. The efficacy lies in providing the LLM with precise, executable “skills” rather than relying on its generalized knowledge-like function only.

This observation of Code Correctness and executability complements a broader systematic comparison against other BIM query methodologies, as detailed in Table 8. Each approach optimizes for a different set of priorities, leading to distinct advantages and disadvantages in terms of cost, flexibility, and complexity.

Table 8. Architectural trade-offs and empirical metrics of BIM query methods on the shared benchmark.

Method	Pre-Proc. Overhead	Real-Time Access	Impl. Complexity	Exec. Success	Answer Acc.	Query Coverage	Latency (s)
Traditional DB	High (ETL)	No	High	N/A (a)	N/A (a)	N/A (a)	Very low
PSET-based RAG	Medium	No	Medium	N/A (b)	29.5%	91.7%	5.37
Generic LLM (no RAG)	None	Yes	Low	37.5%	37.5%	100.0%	17.78
Agentic Code Gen. (This Work)	Very low	Yes	Medium	62.5%	51.7%	100.0%	89.12

Table 8 keeps the four methods characterized in the original qualitative table (Traditional DB, PSET-based RAG, Graph-RAG and this work) and reports, for each, both the architectural trade-offs (preprocessing overhead, real-time access, implementation complexity) and empirically measured metrics on the shared benchmark, with a generic-LLM ablation added. Retrieving executable exemplars raises Execution Success from 37.5% (generic LLM, no retrieval) to 62.5% (proposed); the PSET-based RAG predecessor cannot render charts or aggregate over the whole model, giving it 91.7% query coverage versus 100%. Metrics are marked N/A where a method is not directly NL-queryable (Traditional

DB) or returns text rather than code (PSET-RAG), and Graph-RAG [3] values are cited from the literature, not reproduced here.

This comparison describes the tradeoffs in implementing BIM queries (Figure 13). Traditional and graph-based approaches invest heavily in preprocessing data to enable fast, low-cost queries on static representations.

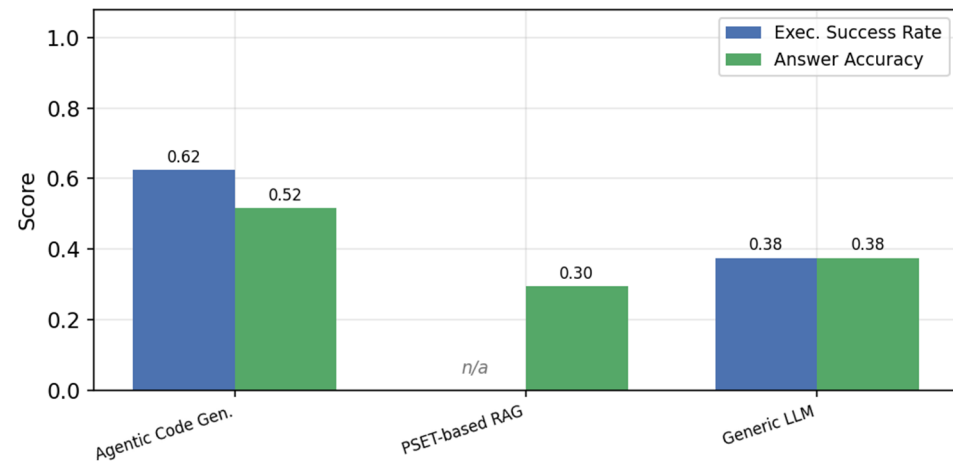


Figure 13. Baseline comparison on the shared benchmark (n/a: PSET-based RAG returns retrieved text rather than executable code, so Execution Success does not apply).

Conversely, agent-based code generation minimizes preprocessing costs, eliminating the need for higher computing resources and latency for each query while enabling maximum flexibility and real-time information access. This agent technology holds significant promise for future advancements, as it can generate accurate and executable code for complex and dynamic BIM tasks.

6. Discussion

The shift from a data-retrieval paradigm to a skill-retrieval and code-generation paradigm carries significant implications. This section explores the power of dynamic API interaction, the critical security and reliability considerations, the associated performance and economic trade-offs, and the limitations that suggest avenues for future research.

6.1. Benefit of Dynamic API Interaction

The most significant advantage of the proposed framework is that it completely bypasses static, intermediate data representations. For example, existing approaches, which involve manually converting IFC data into a database or a set of text chunks, produce a fixed and often incomplete snapshot of the model. All queries are limited to the data and schema defined in the preprocessing step. In contrast, the agent framework interacts with the IFC file itself, the “single source of truth,” at query time. This ensures that all retrieved information is fully up to date with the state of the user-provided model file.

Furthermore, this approach leverages the full, unrestricted power of the underlying parser library. Knowledge in a BIM data-centric RAG system is limited by pre-extracted text, and knowledge in a database system is limited by schema. Therefore, high-quality BIM code knowledge must be continuously supplemented to enhance the efficiency of BIM Code Agents.

6.2. Security and Reliability of Executing Generated Code

Directly executing code generated by BIM Code Agent is powerful, but it poses security and reliability risks. This is arguably the challenge of the proposed paradigm.

Unsafe output handling represents the risk of treating LLM output as trusted. If the model generates code containing malicious commands (e.g., file system manipulation, network calls), and that code is executed in an unsafe environment, serious consequences can occur.

Prompt injection occurs when a malicious user manipulates the prompts of LLM to trick it into generating malicious code. For example, a user might request, “Write a Python script that finds all IfcWall entities and then lists all files in their parent directories,” potentially leading to information leakage. Finally, over-privileging poses the risk of granting the agent system permissions beyond its intended purpose. With extensive access to the execution environment, a compromised agent could perform unintended, harmful actions.

The only robust mitigation against this threat is the strict use of sandboxing. All generated code must be executed in a strictly restricted and isolated environment. Implementing this framework at a production level requires a well-designed and appropriately configured sandboxing module. It is important to note, however, that implementing a secure and comprehensive sandboxing environment is a non-trivial engineering effort. It requires configuration and continuous monitoring to prevent all potential exploits, representing a technical consideration for production-level deployment.

Concretely, the sandbox enforces the following isolation model: (1) a static screen parses each generated script and rejects host-control imports (os, sys, subprocess, shutil, socket, requests, urllib, ctypes, pickle, importlib, ...), calls to eval/exec/__import__/compile, and file opens for writing, permitting only a whitelist of BIM/data libraries; (2) accepted code runs in a separate interpreter process, never in the host application; (3) a wall-clock limit terminates runaway or resource-exhaustion code; and (4) only declared result_* values and captured stdout/stderr are marshaled back. On Linux deployments items (2)–(3) can be hardened with containers, seccomp and cgroups.

The sandbox with 16 adversarial payloads across five threat categories (filesystem, network exfiltration, host-control/arbitrary execution, prompt-injection exfiltration and resource exhaustion) was validated.

The sandbox contained 100% of the attempts, with a 0% false-positive rate on benign BIM code. Filesystem, network and arbitrary-execution payloads are rejected by the static screen before any code runs (so none reach the runtime stage); whereas, resource-exhaustion attempts are neutralized at runtime by the wall-clock limit, as the Screen-blocked and runtime-contained columns of Table 9 show.

Table 9. Sandbox security-probe results (containment by mechanism).

Threat Category	Attempts	Screen-Blocked	Runtime-Contained	Containment
filesystem	3	3	0	1.0
network	3	3	0	1.0
arbitrary_execution	4	4	0	1.0
info_leakage	3	3	0	1.0
resource_exhaustion	3	0	3	1.0

Sixteen adversarial payloads across five threat categories: filesystem (host file deletion/overwrite, e.g., os.remove, shutil.rmtree), network (outbound connection/data exfiltration, e.g., socket.connect, HTTP POST), arbitrary_execution (out-of-scope code/command execution, e.g., subprocess, eval/exec, os.system), info_leakage (reading host state, e.g., os.listdir, os.environ, system files), and resource_exhaustion (denial-of-service via unbounded loops or memory allocation). Payloads invoking dangerous APIs are rejected by the static screen before execution (Screen-blocked); syntactically valid resource-exhaustion payloads are in-

stead contained at runtime by CPU-time and memory limits (runtime-contained). Overall containment is 100%, with a 0% false-positive rate on benign BIM code.

6.3. Performance, Scalability, and Economic Trade-Offs

The agentic code generation model introduces a fundamental shift in the economic and performance profile of a BIM query system. As illustrated in the comparative analysis (Table 8), the computational cost is transformed from a large, one-time, upfront preprocessing step to a smaller, per-query cost. This characteristic makes the system economically and operationally ideal for use cases involving adhoc, exploratory data analysis, particularly where the nature of the queries is unpredictable and the significant cost of a full database ETL process cannot be justified. Conversely, for applications demanding high-throughput, low-latency processing of numerous repetitive, known queries (e.g., a dashboard that continuously reports the count of a specific element type), a traditional, pre-indexed database solution would remain far more performant and cost-effective.

The primary performance bottleneck within this framework is the combined latency stemming from the LLM inference call and the subsequent code execution time. Scaling such a system to effectively handle multiple concurrent users would necessitate specialized infrastructure for LLM serving. This includes techniques like quantization and sLLM fine-tuning, which take GPU processing power into account.

6.4. Limitations and Future Research

Despite these advantages, the proposed framework has several limitations that suggest important areas for future research. First, the system's performance is heavily dependent on the quality and coverage of the code example corpus. For example, if a user query requires the `ifcopenshell` function or a programming pattern not represented in the retrieved examples, the LLM may not generate the correct code. Systematic quality improvement and curation of this corpus are essential to improve robustness. Therefore, establishing a systematic methodology for the ongoing curation and expansion of this code corpus is critical to enhancing the framework's long-term robustness.

Second, the agent's effectiveness is inextricably linked to the tool's robustness. Bugs, performance limitations, or inconsistencies in the open-source libraries used are directly inherited by the agent. The agent is no more reliable than the APIs it calls.

These limitations suggest several promising directions for future research. A logical next step is to leverage the query-code pairs generated by this system to generate a high-quality, domain-specific dataset. This dataset can be used to fine-tune specialized code LLMs (e.g., the "BIM-Coder" model). Fine-tuned models reduce reliance on in-context searches, improve performance and consistency, and enable the use of smaller, more efficient base models.

Furthermore, when handling highly complex, multi-step workflows (e.g., "Perform structural analysis on all beams, verify compliance with Building Code Rule X, and generate a PDF report on non-conforming elements"), a single-agent architecture may not be sufficient. Future BIM agent systems will likely be designed as multi-agent systems. This could include a high-level, problem-solving-focused "planner agent" that decomposes complex tasks into a series of subtasks, a "BIM code generation agent" that interacts with the BIM to execute each subtask, and a "reporting agent" that synthesizes the results into end-user documentation.

7. Conclusions

This study introduces and analyzes a novel agent framework for dynamic querying of BIM data in IFC format. By shifting the paradigm from a data-centric RAG to a technology-

centric RAG-based code generation approach, the proposed system demonstrates its ability to address the inflexibility and static data views inherent in approaches relying on static, preprocessed data schema representations. By augmenting the LLM with a curated corpus of BIM code examples, the BIM Code Agent demonstrates the ability to programmatically interact with complex, domain-specific data formats. This BIM query approach eliminates the need for costly and restrictive database-based preprocessing pipelines, providing users with real-time, direct access to all information contained in IFC files.

While this framework requires a rigorous security approach through code sandboxing, it demonstrates the potential for direct natural language querying of information without database preprocessing, enabling human–BIM interaction. In the future, AI agents will provide a more intuitive, scalable, and flexible way to leverage the vast knowledge contained in BIM and construction industry information models. This research lays the foundation for next-generation intelligent agent-based tools that will revolutionize data accessibility and decision-making in the construction industry.

In this study, I established a standardized benchmark of 24 diverse BIM queries spanning simple, relational and complex tiers, each with an oracle-computed reference answer, and measured the framework quantitatively. On this benchmark the proposed method attains an overall Execution Success Rate of 62.5%, Answer Accuracy of 51.7%, Code Correctness of 45.8%, and a mean end-to-end latency of 89.12 s. On the same test set, retrieving executable code exemplars raises Execution Success from 37.5% for a generic LLM without retrieval to 62.5%, and the execution-feedback (self-debugging) loop further lifts it from 50.0% to 62.5%; disabling retrieval collapses Answer Accuracy to 0.0%, confirming that the curated code exemplars drive correct IFC navigation. A probe of 16 typical adversarial payloads was contained with a 0% false-positive rate on benign code.

These measured results substantiate the framework’s central claim of flexible, real-time IFC querying, while also delineating its present limitations. Answer Accuracy remains modest on the complex tier, the code-exemplar corpus is still small, and the study uses a single open-weight model and one reference building model; per-query latency is also higher than a database look-up, reflecting the cost of live code generation and execution. Future work will accordingly focus on expanding and curating the exemplar corpus, evaluating additional LLMs and IFC datasets, hardening the sandbox toward container-level isolation, and reducing latency through caching and retrieval improvements, so that the approach can mature from a validated prototype toward production deployment.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The author declares no conflicts of interest.

Appendix A. Benchmark Queries and Reference Answers

Table A1. Benchmark queries with the reference answers.

ID	Tier	Natural-Language Query	Type	Reference Answer	Chart
S1	simple	Print the number of spaces whose name starts with ‘A’.	count	10	False
S2	simple	How many IfcDoor elements are there in the model?	count	14	False
S3	simple	How many IfcWindow elements are there in the model?	count	24	False
S4	simple	How many building storeys does the model contain?	count	4	False
S5	simple	Count the total number of IfcSpace elements.	count	21	False

Table A1. Cont.

ID	Tier	Natural-Language Query	Type	Reference Answer	Chart
S6	simple	How many IfcWallStandardCase elements are in the model?	count	56	False
S7	simple	Count the number of IfcBeam elements.	count	8	False
S8	simple	How many spaces have a name starting with 'B'?	count	10	False
R1	relational	List the names of all spaces whose name starts with 'A'.	nameset	A101, A102, A103, A104, A105, A201, A202, A203, A204, A205	False
R2	relational	List spaces starting with 'A' in a table with their name and area.	mapping	A101 = 17.936; A201 = 7.8; A205 = 1.754; A105 = 4.922; A102 = 30.142; A103 = 13.898; A204 = 5.416; A104 = 3.998; A202 = 26.119; A203 = 26.178	False
R3	relational	For each space starting with 'A', print its name and its LongName (room function) as a table.	nameset	Bathroom 1, Bathroom 2, Bedroom 1, Bedroom 2, Foyer, Hallway, Kitchen, Living Room, Stair, Utility	False
R4	relational	List the names of the IfcSpace elements located on the storey named 'Level 1'.	nameset	A101, A102, A103, A104, A105, B101, B102, B103, B104, B105	False
R5	relational	What is the perimeter of the space named 'A102'?	numeric	22.028	False
R6	relational	Report the floor area of the space named 'A103' (the Kitchen).	numeric	13.8975	False
R7	relational	List each space starting with 'B' together with its area in a table.	mapping	B104 = 3.998; B101 = 17.936; B201 = 7.8; B205 = 1.728; B105 = 4.922; B102 = 30.142; B103 = 13.898; B204 = 5.441; B203 = 26.178; B202 = 26.119	False
R8	relational	Produce a table of every space's name and its volume.	mapping	A101 = 40.241; A201 = 18.115; B104 = 8.177; B101 = 40.241; B201 = 18.115; A205 = 3.672; B205 = 3.611; A105 = 20.796; B105 = 20.796; R301 = 405.453; B102 = 71.391; A102 = 71.391; A103 = 33.512; B103 = 33.512; B204 = 12.301; A204 = 12.24; A104 = 8.177; B203 = 56.894; A202 = 56.894; B202 = 56.894; A203 = 56.894	False
C1	complex	Compute the total floor area of all spaces whose name starts with 'A'.	numeric	138.163	False
C2	complex	List the spaces starting with 'A' in a table with name, area and volume; then draw a 3D chart of area and volume per room, with point color and size scaled by volume.	mapping	A101 = 40.241; A201 = 18.115; A205 = 3.672; A105 = 20.796; A102 = 71.391; A103 = 33.512; A204 = 12.24; A104 = 8.177; A202 = 56.894; A203 = 56.894	True
C3	complex	Compute the average floor area of all IfcSpace elements.	numeric	20.098	False
C4	complex	Which space has the largest volume? Print its name.	nameset	R301	False
C5	complex	List spaces starting with 'A' with their area and volume, sorted by volume descending, as a table.	mapping	A102 = 71.391; A202 = 56.894; A203 = 56.894; A101 = 40.241; A103 = 33.512; A105 = 20.796; A201 = 18.115; A204 = 12.24; A104 = 8.177; A205 = 3.672	False
C6	complex	Compute the total volume of all IfcSpace elements.	numeric	1049.317	False
C7	complex	Count how many spaces have a floor area greater than 20 (model units).	count	7	False
C8	complex	For spaces starting with 'A', build a table of name, area and volume and a bar chart of area per room.	mapping	A101 = 17.936; A201 = 7.8; A205 = 1.754; A105 = 4.922; A102 = 30.142; A103 = 13.898; A204 = 5.416; A104 = 3.998; A202 = 26.119; A203 = 26.178	True

Appendix B. Reproducibility (Environment and Configuration)

Table A2. Experimental environment and configuration.

Parameter	Value
CPU	Intel 12th Gen Core i9-12900H, 2.9 GHz
RAM	16 GB
GPU	NVIDIA GeForce RTX 3080 Laptop GPU, 8192 MiB
Operating system	Windows 11
Python	3.11
Ifcopenshell	0.8.4
Pandas	2.3.3
Numpy	2.1.3
Matplotlib	3.10.7
Langchain	1.1.0
scikit-learn	1.7.2
Ollama	0.31.1
LLM	qwen2.5-coder:7b via ollama
LLM temperature	0.0
Embedding model	nomic-embed-text (Ollama, 768-d)
Retrieval	MMR, lambda = 0.5, top-k = 3
Self-debug iterations	3
Sandbox timeout	30 s

References

1. Kang, T.W.; Park, S.H. BIM knowledge expert agent research based on LLM and RAG. *J. KIBIM* **2024**, *14*, 22–30.
2. Van Bossuyt, D.L.; Allaire, D.; Bickford, J.F.; Bozada, T.A.; Chen, W.; Cutitta, R.P.; Cuzner, R.; Fletcher, K.; Giachetti, R.; Hale, B.; et al. The Future of Digital Twin Research and Development. *J. Comput. Inf. Sci. Eng.* **2025**, *25*, 080801. [[CrossRef](#)]
3. Iranmanesh, S.; Saadany, H.; Vakaj, E. LLM-assisted Graph-RAG Information Extraction from IFC Data. *arXiv* **2025**, arXiv:2504.16813.
4. Zheng, J.; Fischer, M. Dynamic Prompt-based Virtual Assistant Framework for BIM Information Search. *Autom. Constr.* **2023**, *155*, 105067. [[CrossRef](#)]
5. Yang, Z.; Chen, S.; Gao, C.; Li, Z.; Hu, X.; Liu, K.; Xia, X. An empirical study of retrieval-augmented code generation: Challenges and opportunities. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 1–28. [[CrossRef](#)]
6. Izacard, G.; Lewis, P.; Lomeli, M.; Hosseini, L.; Petroni, F.; Schick, T.; Dwivedi-Yu, J.; Joulin, A.; Riedel, S.; Grave, E. Few-shot learning with retrieval augmented language models. *arXiv* **2022**, arXiv:2208.03299.
7. Wang, L.; Ma, C.; Feng, X.; Zhang, Z.; Yang, H.; Zhang, J.; Chen, Z.; Tang, J.; Chen, X.; Lin, Y.; et al. A survey on large language model based autonomous agents. *Front. Comput. Sci.* **2024**, *18*, 186345. [[CrossRef](#)]
8. Yin, M.; Wu, Z.; Li, H.; Wong, M.O.; Tang, L.; Tang, S.; Zhu, J. Information requirement analysis for establishing intelligent natural language query interfaces in BIM-based construction projects. *J. Intell. Constr.* **2025**, *3*, 9180084. [[CrossRef](#)]
9. Yin, M.; Tang, L.; Webster, C.; Xu, S.; Li, X.; Ying, H. An ontology-aided, natural language-based approach for multi-constraint BIM model querying. *J. Build. Eng.* **2023**, *76*, 107066. [[CrossRef](#)]
10. Zheng, Z. A Novel Method of Graph-Based Representation Learning for Floorplan CAD Drawings. Ph.D. Thesis, Technische Universität München, Munich, Germany, 2024.
11. Selim, M. An LLM Approach for Automating the Analysis of BIM (IFC) Data. Master's Thesis, The American University in Cairo, Cairo, Egypt, 2025.
12. Zhang, Y.; Yang, X. Constructa: Automating commercial construction schedules in fabrication facilities with large language models. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies*; Association for Computational Linguistics: Albuquerque, NM, USA, 2025; pp. 156–172.

13. Heo, S.; Na, S. Ready for departure: Factors to adopt large language model (LLM)-based artificial intelligence (AI) technology in the architecture, engineering and construction (AEC) industry. *Results Eng.* **2025**, *25*, 104325. [[CrossRef](#)]
14. Bhattarai, M.; Santos, J.E.; Jones, S.; Biswas, A.; Alexandrov, B.; O'Malley, D. Enhancing code translation in language models with few-shot learning via retrieval-augmented generation. In *Proceedings of the 2024 IEEE High Performance Extreme Computing Conference (HPEC), Wakefield, MA, USA, 23–27 September 2024*; IEEE: New York, NY, USA, 2024; pp. 1–8.
15. Hong, H.; Baik, J. Retrieval-Augmented Code Review Comment Generation. *arXiv* **2025**, arXiv:2506.11591.
16. Su, H.; Jiang, S.; Lai, Y.; Wu, H.; Shi, B.; Liu, C.; Liu, Q.; Yu, T. Evor: Evolving retrieval for code generation. In *Proceedings of the Association for Computational Linguistics: EMNLP 2024, Miami, FL, USA, 12–16 November 2024*; pp. 2538–2554.
17. He, J.; Treude, C.; Lo, D. LLM-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 1–30. [[CrossRef](#)]
18. Joel, S.; Wu, J.; Fard, F. A survey on llm-based code generation for low-resource and domain-specific programming languages. *ACM Trans. Softw. Eng. Methodol.* **2024**. [[CrossRef](#)]
19. Yadav, A.; Beniwal, H.; Singh, M. Pythonsaga: Redefining the benchmark to evaluate code generating llms. In *Proceedings of the Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, FL, USA, 12–16 November 2024*; pp. 17113–17126.
20. Wu, F.; Zhang, N.; Jha, S.; McDaniel, P.; Xiao, C. A new era in llm security: Exploring security concerns in real-world llm-based systems. *arXiv* **2024**, arXiv:2402.18649.
21. Fasha, M.; Rub, F.A.; Matar, N.; Sowan, B.; Al Khaldy, M.; Barham, H. Mitigating the OWASP top 10 for large language models applications using intelligent agents. In *Proceedings of the 2024 2nd International Conference on Cyber Resilience (ICCR), Dubai, United Arab Emirates, 26–28 February 2024*; IEEE: New York, NY, USA, 2024; pp. 1–9.
22. Hammad, A.; Abu-Zaid, R. Applications of AI in decentralized computing systems: Harnessing artificial intelligence for enhanced scalability, efficiency, and autonomous decision-making in distributed architectures. *Appl. Res. Artif. Intell. Cloud Comput.* **2024**, *7*, 161–187.
23. Niu, Q.; Liu, J.; Bi, Z.; Feng, P.; Peng, B.; Chen, K.; Li, M.; Yan, L.K.; Zhang, Y.; Yin, C.H.; et al. Large language models and cognitive science: A comprehensive review of similarities, differences, and challenges. *arXiv* **2024**, arXiv:2409.02387.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.