www.itcon.org - Journal of Information Technology in Construction - ISSN 1874-4753

INTEGRATING LARGE LANGUAGE MODELS AND KNOWLEDGE GRAPHS FOR ADAPTIVE DESIGN REVIEW

SUBMITTED: May 2026
PUBLISHED: August 2026
EDITOR: Robert Amor
DOI: 10.36680/j.itcon.2026.038

Maen Alnuzha, Ph.D. candidate
Faculty of Civil and Environmental Engineering, Technion Israel Institute of Technology, Israel
https://orcid.org/0009-0004-4617-2905
maen@campus.technion.ac.il

Tanya Bloch, Dr
The Rosalinde and Arthur Gilbert Foundation-Krengel Family Faculty Fellow, Faculty of Civil and Environmental Engineering, Technion Israel Institute of Technology, Israel
https://orcid.org/0000-0002-3588-9685
bloch@technion.ac.il

SUMMARY: *Automated Compliance Checking (ACC) systems are fundamentally static, unable to easily adapt to new regulations, project constraints, organizational, or practitioner-defined rules. This paper presents a framework integrating Knowledge Graphs (KGs) and Large Language Models (LLMs) to support a more extensible design review environment. In this framework, the KG acts as a structured repository for rules and executable logic, while the LLM serves as an intelligent interface. The central innovation is the human-in-the-loop feedback mechanism, where new logic generated by the LLM is validated, executed, and permanently stored in the KG, transforming it into an active, evolving validation engine. Following a Design Science Research (DSR) methodology, we implement and evaluate the framework as a prototype embedded as an Autodesk Revit add-in, demonstrating its ability to retrieve and execute existing rules from the KG, capture new requests during design, and maintain a verifiable, adaptive compliance checking system. Across a two-experiment evaluation, the system achieved 100% mapping accuracy for six existing rules, while generating new executable rules from natural language succeeded in 70% of 20 trials. Performance was strong on parameter-based checks (100%) but dropped on rules involving spatial reasoning (20–60%), where the LLM still struggles to produce reliable logic.*

KEYWORDS: *Automated Code Compliance (ACC), Machine Learning (ML), rule-based checking, Building Information Modeling (BIM), Knowledge Graph (KG), Large Language Models (LLMs).*

REFERENCE: *Alnuzha, M., & Bloch, T. (2026). Integrating large language models and knowledge graphs for adaptive design review. Journal of Information Technology in Construction (ITcon), 31, 888-916. https://doi.org/10.36680/j.itcon.2026.038*

1. INTRODUCTION

During the design phase, designers must validate their work against numerous codes and standards (Eastman et al. 2009). The field of Automated Compliance Checking (ACC) has been an active area of research for decades, aiming to automate and streamline this process (Amor and Dimyadi 2021). Despite this long-standing effort, the full potential of design review remains unrealized, hindered by a series of deep-seated challenges (Amor and Dimyadi 2021; Beach et al. 2020).

Previous ACC research focused on two primary challenges. The first is rule interpretation, which involves translating ambiguous, human-readable regulations into machine-executable rules (Zhang and El-Gohary 2016b; Zhou and El-Gohary 2014), and the second is model preparation, the process of semantically enriching BIM data to ensure it is "check-ready" (Bloch and Sacks 2018; Dinis et al. 2022; Wang et al. 2022). While significant progress has been made within each research stream, the proposed solutions remain limited, focused on specific codes and struggle to scale to a wider range of regulations (Alnuzha and Bloch 2025).

As noted in the work of Bloch et al. (2019) and Sacks et al. (2020), the core technology of commercial ACC systems has not fundamentally changed over the years. Current ACC systems are built on inflexible, hard-coded, rule-based mechanisms, and authoring new rules typically require proficiency in specific programming languages (Lee et al. 2020). This technical barrier limits the ability of such systems to evolve with changing regulations and design practices.

Traditionally, design review is a post-design workflow, meaning that issues are discovered late in the process, which can lead to costly rework and disrupt the design flow (Chen and Bao 2025). This has motivated a shift in research towards continuous compliance, aiming to create a more dynamic and responsive design environment (Izbash and Babayev 2024; Li and Maiti 2025). However, an interactive design environment that provides immediate feedback during the design process has not been developed yet.

Beyond these technological and workflow limitations, regulations represent only the minimal requirements and do not always reflect best practice (Edwards et al. 2019). Project-specific constraints, company standards, and rules adopted from professional practice are routinely applied alongside published code (Fitkau and Hartmann 2024). However, ACC systems are designed around formal regulatory documents and provide no clear mechanism for considering such rules. A design review system that can be extended with new rules during use, regardless of where those rules originate, would provide stronger support to designers.

To address these limitations, this paper presents a framework that allows designers to easily extend the rule library themselves during design. Relying on the capability of a Large Language Model (LLM) to generate executable code, the framework provides designers with an interface to express new rules in natural language. The role of the Knowledge Graph is to serve as a persistent storage layer and provide a structured representation of building elements, properties, and regulatory context, allowing the system to retrieve relevant information that focuses the LLM on the specific domain of the request.

The rest of this paper is structured as follows: Section 2 presents the state of the art of the design review process, examining the historical challenges of ACC and the specific roles of KGs and LLMs in this context. Section 3 outlines the research aims and methodology. Section 4 presents the conceptual framework description and system implementation. Section 5 presents the results of experiments designed to test and evaluate the proposed framework. The results are discussed in Section 6, where we analyze our findings, the challenges of managing LLMs, the potential of our approach, and outline the limitations of the current study and directions for future work. Finally, Section 7 concludes the paper by summarizing its contributions.

2. BACKGROUND

The design review process serves as the primary mechanism for ensuring quality, correctness, and compliance of a developed design. The long-standing vision of automating this process dates back decades (Eastman 1975). Over time, this vision led to the formalization of the ACC field in both research and commercial practice.

Established ACC platforms, such as Solibri (2025) and CORENET (Khemlani 2005), can be characterized as "black box" solutions that rely on hard-coded, proprietary logic inaccessible to the end-user (Ismail et al. 2023). This rigidity creates a critical bottleneck when regulations change and require modification of existing executable rules (Dimyadi et al. 2016). In response, research has moved towards more transparent "white box" methodologies,

such as tools built on Visual Programming Languages (VPLs) (Preidel and Borrmann 2015, 2016, 2018) which enhance usability for non-programmers.

In parallel, substantial advances have been made in rule interpretation, ranging from rule markup methods such as RASE (Requirement, Applicability, Selection, Exception) (Hjelseth and Nisbet 2011), Natural Language Processing (NLP), Machine Learning (ML) (Zhang and El-Gohary 2016a, 2015, 2016b), and recently, LLM-based extraction of regulatory requirements (Zhu et al. 2024).

Despite this progress, existing systems lack a general mechanism for incorporating new rules into the checking process once the system has been deployed. Whether a new rule originates from a revised regulation, a project-specific requirement, a company standard, or a design rule of thumb adopted from professional practice, the path to making it executable typically still requires offline rule authoring by developers or technical specialists. For example, extending tools such as Solibri beyond their built-in rule templates typically requires interaction with the Solibri API and the programming skills that come with it. Equally absent is a mechanism for addressing rules that are not expressed in regulatory documents. Rules are often implemented by designers for a single project or compliance task, but do not accumulate into a shared resource that is discoverable and executable by other practitioners on future projects. The rule base does not evolve through use, and the knowledge invested in each new rule is not retained for future reuse across projects. This limits the ability of current systems to adapt to requirements that emerge during the design process.

2.1 The use of Knowledge Graphs for representing regulatory knowledge

Knowledge Graphs (KGs) are formal, graph-based representations where entities, concepts, and their properties are represented as nodes, and the relationships between them are represented as edges, forming a network of subject-predicate-object triples (Zhang et al. 2018). In ACC research, knowledge graphs are frequently applied to organize building concepts, elements, and the regulatory rules that link them together (Pauwels et al. 2017; Peng and Liu 2023). These graphs offer a way to express this knowledge in a structured form rather than as hard-coded checking logic (Zhu et al. 2025).

Building regulations are written in human language, but building information models are composed of geometry, parameters, and relational data. A knowledge graph provides a representation that sits between these two formats. It captures the building domain at a conceptual level by including entities and their properties in a form that can be queried and traversed (Wang and El-Gohary 2023). Through this representation, regulatory knowledge can be aligned with building data and queried using languages such as SPARQL. These languages can execute checks and infer new information by traversing the graph (Ma et al. 2024; Zheng et al. 2022).

A KG also enables abstraction across element types that share the same regulatory treatment, despite differing in form and function. For example, regulations governing means of egress apply to doors, stairs, corridors, ramps, and exit passageways. These are very different types of elements, but the regulation treats them under shared requirements, such as minimum clear width and continuity of the egress path (International Code Council 2021). A KG can link these elements to a shared concept, such as MeansOfEgressComponent, so that a rule defined against the shared concept applies to all of them without requiring separate code for each element type. This kind of organization becomes more important as the rule base grows. A flat library of code files can become difficult to navigate. A KG instead organizes rules through their connections to building elements and regulations. This supports retrieval that is informed by the structure of the domain rather than by simple keyword matching (Wang and El-Gohary 2023; Zheng et al. 2025a).

Translating regulations into their KG representations is not an automated process (Zheng et al. 2025a). As detailed in the literature, it typically involves knowledge modeling and knowledge population. The first stage is to define an ontology that establishes the shared vocabulary and constraints for a specific domain (Chen and Luo 2016). The second stage is populating the graph with instances of knowledge extracted from regulatory documents. This often uses NLP techniques to identify entities and extract their properties (Peng and Liu 2023; Wang and El-Gohary 2023). Domain-specific ontologies have been developed for areas such as fire safety (Fitkau and Hartmann 2024) and model quality checking (Ma et al. 2024). The lack of training data has motivated the creation of foundational datasets such as the CODE-ACCORD corpus (Hettiarachchi et al. 2025). This is an output of the European ACCORD project (2025) and consists of regulatory sentences annotated by experts.

Across these research efforts, the focus is on extracting knowledge from formal regulatory documents. The knowledge graph is usually populated through offline pipelines managed by technical specialists. The resulting graph is typically treated as a static asset that is built once and revised only periodically (Yang et al. 2023; Zhu et al. 2025). This update cycle limits the ability of the rule base to evolve while the system is being used. Current systems do not provide a mechanism for adding new rules that arise during the design process.

2.2 Large Language Models for ACC

Unlike traditional NLP or ML models that require extensive, task-specific engineering and large annotated datasets (Alnuzha and Bloch 2025; He et al. 2025; Zhong et al. 2020), LLMs like GPT, Llama, and Claude are pre-trained on vast and diverse corpora of text and code. This pre-training provides broad capabilities for natural language understanding, reasoning, and generation, which can be adapted to specific domains with minimal data, through in-context or few-shot learning (Brown et al. 2020; Joffe et al. 2025; Touvron et al. 2023). In the ACC domain, the capabilities of LLMs to interpret regulations (Joffe et al. 2025) and generate executable code (Madireddy et al. 2025; Ying and Sacks 2024) are transformative.

LLMs can also function as a "virtual assistant" for querying BIM, classifying user intent, and identifying the correct parameters for a database query from a simple natural language question (Zheng and Fischer 2023). This is achieved by designing dynamic prompts that provide the LLM with the necessary context about the available data structures, enabling it to map a user's question like "How high is the door?" to the correct entity ("door") and attribute ("height"). This approach has been applied to various domains, including construction quality checking (He et al. 2025), safety regulation extraction (Tran et al. 2024), and general code compliance queries (Joffe et al. 2025). However, LLMs are trained primarily on text and tend to struggle with the spatial and topological reasoning required to interact with building models, which has motivated proposals to support them with graph-based representations of the BIM model (Du et al. 2026).

The reliability of LLMs in domain-specific applications can be improved by supplementing the model's internal knowledge with relevant external information (He et al. 2025; Isaev et al. 2023). This is usually accomplished through a technique called Retrieval-Augmented Generation (RAG) (Lewis et al. 2020; Wan et al. 2025). In a RAG framework, when a user poses a query, the system first retrieves the most relevant text snippets from a corpus of source documents before prompting the LLM. The LLM then uses this retrieved context to generate a more accurate response, mitigating the risk of "hallucination" or fabricating incorrect information (He et al. 2025; Joffe et al. 2025). Some approaches enhance this retrieval step with hybrid search methods, combining traditional keyword-based search (e.g., TF-IDF) with vector search to better handle the domain-specific terminology found in AEC regulations (He et al. 2025). This RAG-based interpretation allows LLMs to reason over specific, up-to-date document sets without the need for constant, costly fine-tuning.

The generation capability of LLMs opens the possibility of translating a user's requirement directly into an executable format. Recent work has shown that LLMs can produce functional Python scripts for checking compliance rules within a BIM environment, such as Revit (Madireddy et al. 2025). However, they report that their workflow still requires manual intervention for script execution within Revit and for correcting errors. Iversen and Huang (2026) proposed an end-to-end artifact in which an LLM performs rule interpretation, model preparation, execution, and reporting as a single workflow, evaluated on 130 clauses from the Norwegian TEK17 regulation. Their evaluation showed that an LLM can carry out these tasks without requiring additional information to be added to the BIM model beyond what is normally available. The authors note, however, that the artifact handles only geometric requirements, and that non-geometric design intent, such as the intended use of a room, still needs to be entered by the designer. In this case, their artifact depends on a manually developed library of extraction scripts, does not currently distinguish data insufficiency from non-compliance, and re-runs the full LLM pipeline for every check rather than reusing previously interpreted logic. Once a compliance check is performed, the interpretation, generated script, or reasoning trace is not captured in a form that can be reused on subsequent projects without re-invoking the LLM.

A more advanced application involves using an LLM to map complex regulatory clauses to a database of predefined "atomic functions," effectively deconstructing a complex rule into a series of executable function calls (Zheng et al. 2025b). This moves beyond simple parameter checks to handle rules with implicit properties and complex computational logic. For instance, consider a common requirement: "The number of safety exits in a fire protection zone shall not be less than 2." A simple parameter check is insufficient here, as a BIM model typically

does not contain a direct attribute for the 'number of safety exits'. Instead, this is an implicit property that requires complex computational logic to derive. The framework proposed by Zheng et al. (2025b) addresses this by maintaining a library of high-level "atomic functions" such as hasElement(space, element_type) and getProperty(element, property_name). When presented with the rule, the LLM's task is not to write code from scratch, but to correctly identify that these specific atomic functions are the necessary building blocks for the check. This process intelligently maps the ambiguous natural language requirement to a series of verifiable, executable functions, demonstrating a significant leap beyond simple parameter validation. While this structured approach significantly enhances reliability, the system's ability to interpret new rules is contingent on the pre-existing library of atomic functions.

The capabilities of interpretation and code generation have contributed to the development of agentic AI systems for AEC tasks. In an agentic framing, the LLM acts as a reasoning engine that plans tasks, calls external tools to perform actions it cannot do reliably on its own, and coordinates with other agents when a task requires multiple specialized capabilities (Guo et al. 2024; Wang et al. 2024). Foundational work in this direction includes Program-Aided Language models, in which the LLM produces code that is executed externally rather than relying on the model's own arithmetic or logic (Gao et al. 2023), and frameworks that enable LLMs to ask for clarification when a request is ambiguous (Mu et al. 2024). Multi-agent architectures, in which specialized agents coordinate to decompose and solve complex tasks, have been explored in ACC (Ying and Sacks 2024), and in the collaborative nature of architectural design (Tang et al. 2025). While this represents a shift from automated systems that follow pre-programmed instructions to autonomous systems that can reason and act in novel situations, current agentic approaches in ACC do not provide a mechanism to consolidate successful outputs into a persistent validated knowledge, so each new compliance check still depends on re-invoking the LLM rather than on verified rules accumulated through prior use.

2.3 The synergy between KG and LLM

Despite the powerful capabilities of LLMs, their lack of domain-specific grounding and absence of a persistent, verifiable memory make them prone to "hallucinations" and factual inaccuracies (Minaee et al. 2024; Zhao et al. 2023). Without a structured context to anchor their reasoning, their outputs can be inconsistent, untrustworthy, and difficult to retain as persistent knowledge assets.

These limitations mirror, in reverse, the challenges faced by Knowledge Graphs. As previously discussed, KGs provide the structured, explicit, and verifiable knowledge base that LLMs lack. However, they have historically been treated as passive and static repositories whose labor-intensive creation and upkeep limit their ability to reflect evolving user expertise (Yang et al. 2023).

Recognizing these complementary strengths and weaknesses, KG-based RAG has become a primary strategy to alleviate the issues of the LLM's “hallucinations” (Li et al. 2024). LLMs can be used as intelligent front-ends that can translate a user's natural language question into a formal graph query (e.g., Cypher), retrieve the results, and then synthesize them into a human-readable answer (Nakhaee et al. 2024). For example, Zheng et al. (2025a) proposed a "GraphRAG" approach where the retrieval is not limited to simple text chunks but involves traversing the graph to extract specific, interconnected subgraphs. This provides the LLM with a much richer, entity-aware context that better reflects the nested and conditional logic inherent in regulations. In such a synergy, the KG provides the verifiable "source of truth," while the LLM acts as the dynamic, intelligent interface that makes this knowledge accessible and useful.

Feng et al. (2025) combined a regulatory knowledge graph with a fine-tuned LLM-based IfcOpenShell code generator to detect non-compliant values in an IFC model and automatically correct them. In their work, the KG stores semantic regulatory data, and when non-compliant elements are detected, the LLM generates IfcOpenShell code that updates the IFC model values accordingly. The Regulatory Knowledge Graph itself is constructed once through manual annotation and Named Entity Recognition (NER) training over a fixed set of regulatory documents, and the framework does not include a mechanism to extend it during operation. As acknowledged by the authors, the framework is also limited in handling multi-layered rules with attribute dependencies, in covering regulations beyond those used during construction, and in operating inside a BIM authoring environment rather than as a separate offline pipeline.

Zhang et al. (2026) propose a four-module pipeline that uses BERT-based semantic extraction, context-free grammar validation, and human-in-the-loop review of the extracted semantic roles to translate Chinese regulatory clauses into SHACL constraints, with sentence-embedding similarity detecting regulatory changes and a RAG component translating them into updated SHACL constraints. The resulting constraints are validated against IFC models by an off-the-shelf SHACL engine. The authors observed that their approach faces representational limits, with extraction accuracy decreasing on rules involving nested conditional logic, and that rules referring to knowledge outside the regulatory text, such as implicit technical principles, are less well handled. In both cases, the rule base is built around regulatory text as the primary input source, finalized in advance of compliance checking, and operates as an offline production pipeline.

Existing systems typically lack an interactive mechanism for users to modify the knowledge base during operation and remain restricted to regulatory documents. They are largely limited to the rules defined during the development phase and do not easily adapt to requirements that emerge during the design process, whether from revised regulations, project-specific needs, or other sources. The direction this paper investigates is the use of an LLM as an interface that allows designers to extend the KG with new rules after deployment, while preserving the deterministic reuse that KG persistence provides.

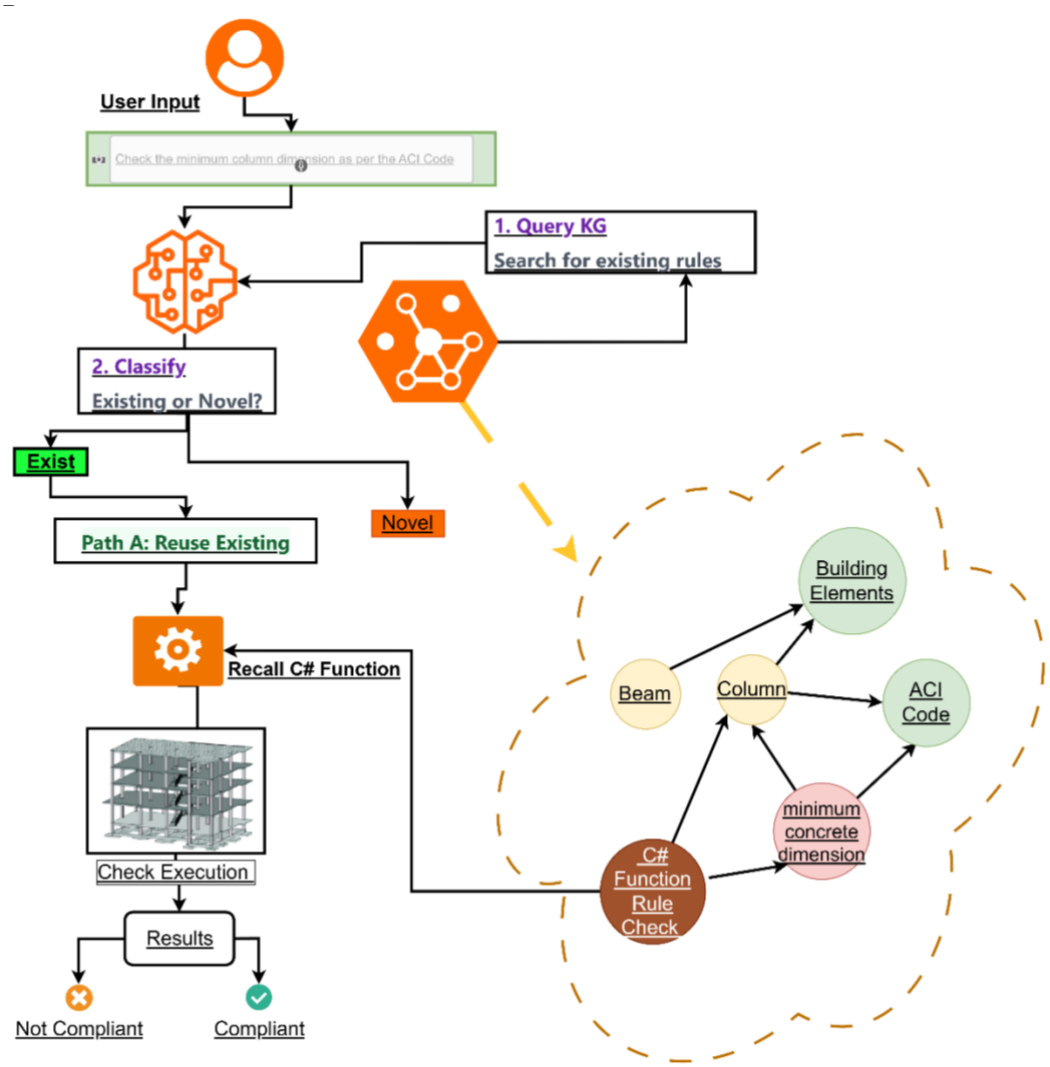

Figure 1: Path A workflow—Reusing existing C# logic stored in the KG.

3. AIMS AND METHODOLOGY

We aim to move beyond the limitations of current ACC tools, which are confined to static, hard-coded rule libraries and lack adaptability to evolving requirements, by proposing a framework that enables designers to perform custom validation checks dynamically, without the need for predefined rule sets or external scripting.

To achieve this goal, the study adopts the Design Science Research (DSR) methodology (Hevner et al. 2004). The primary research artifact of this work is a design review framework that integrates KGs and LLMs to enable flexible and extensible design review directly within the design environment. The framework allows designers to express requirements in natural language, whether derived from formal building codes, organizational standards, project-specific considerations, or other rules they wish to apply during design. A key component of the framework is the KG, which serves as a formal and consistent foundation of executable constraints. It functions as a repository that can be extended over time with new rules validated through user interaction. Any user request that is not represented in the KG is translated into executable logic using LLM capabilities, validated through expert interaction, and persistently stored in the KG. In this way, we move towards a flexible design review environment in which the rule base is not fixed at deployment but grows through user interaction, while preserving the deterministic execution of validated rules.

The proposed framework was implemented as a prototype add-in for Autodesk Revit (Autodesk 2024) and evaluated through a two-stage experimental sequence. The first experiment evaluates the ability to retrieve regulatory knowledge from the KG through LLMs and execute it within a BIM environment. The subsequent experiment validates the complete feedback loop, testing the framework's capacity to capture previously unsupported requirements, generate executable validation logic, and update the KG accordingly. The design of the conceptual framework is described in the next section of this work. Experiment settings and results are presented in section 5.

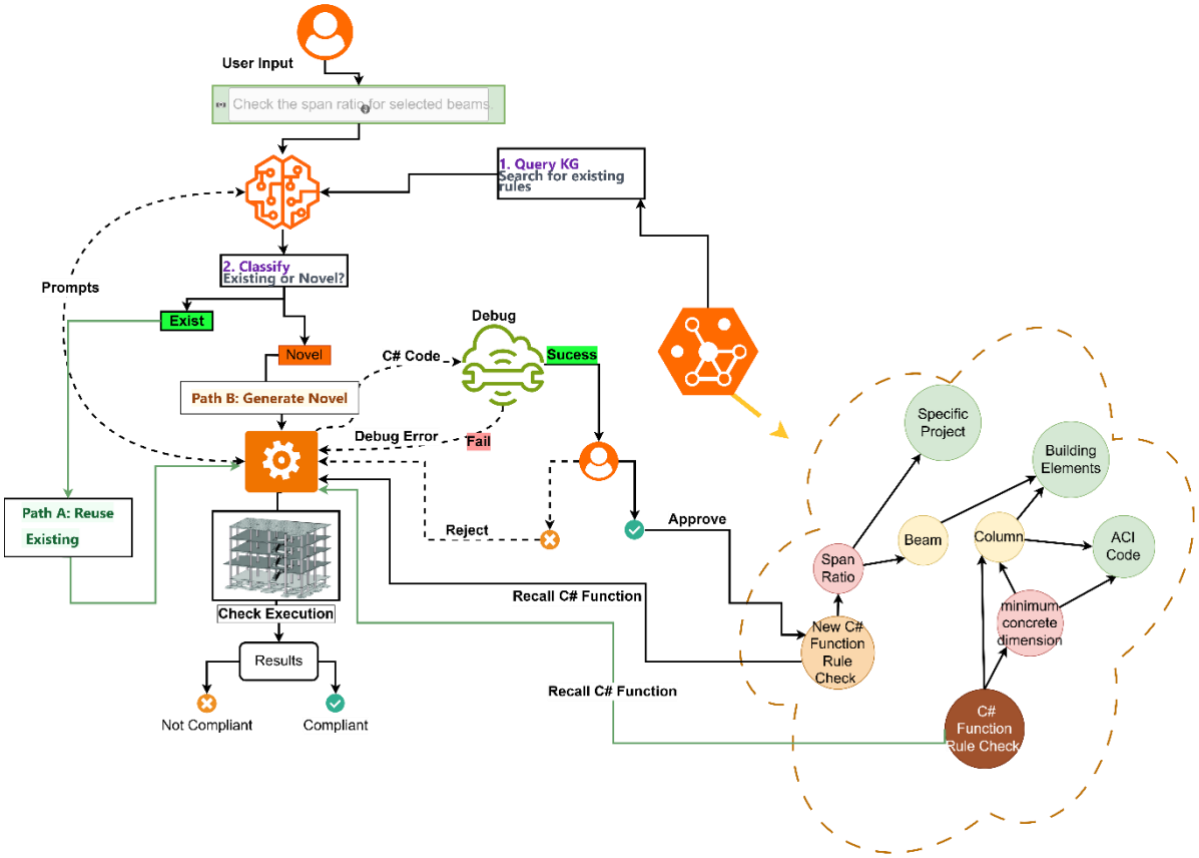

Figure 2: Path B workflow—Generating new C# logic for validation by the user and expanding the KG accordingly.

4. CONCEPTUAL FRAMEWORK DESCRIPTION

The framework is built upon the integration of KG, which provides a structured representation of building regulations, design requirements, and their relationships to building elements, and LLM, responsible for handling new requests. Depending on the nature of the user request, the workflow follows two main paths, as shown in Figures 1 and 2. When a user request matches a rule already defined in the KG (Path A), the framework retrieves the associated C# logic stored alongside the rule and executes it directly, without involving the LLM at execution time. When the user request is novel, and no matching rule exists in the KG (Path B), the LLM generates new executable C# code. The generated code is then compiled and executed, and upon user approval, is stored in the knowledge graph for subsequent reuse.

4.1 System implementation

The technical implementation is a functional C# prototype developed as an add-in for Autodesk Revit. Revit was chosen as the prototype environment, given its widespread use in practice. The system is built using a modular agentic architecture that supports multiple compliance domains through shared and specialized components. The designer selects the desired compliance type (architectural or structural) through the Revit add-in interface, which activates the corresponding domain-specific module. Both modules share core infrastructure components, as illustrated in Table 1, while maintaining separate event handlers and prompt managers tailored to their respective domains.

Table 1: Primary prototype components.

Component	Responsibility
AiComplianceEventHandler	Serves as the main orchestrator, coordinating the workflow from user input to final feedback. Manages request classification, invokes appropriate services based on classification results, and handles the self-correction loop when generated code fails to compile. Domain-specific implementations exist for architectural compliance (GeneralArchitectAiComplianceEventHandler) and structural compliance (StructuralAiComplianceEventHandler).
AiCommsManager	Responsible for constructing prompts, sending requests to the LLM, and parsing the structured JSON and C# code responses. Each compliance domain maintains its own implementation with domain-specific prompt templates (e.g., AiCommsManager_ArchitecturalCompliance for architectural checks).
KnowledgeGraph	Built on the dotNetRDF library (Vesse et al. 2025), this shared component manages the system's semantic memory. It handles persistence and querying of all compliance rules as semantic triples stored in Turtle format, maintaining concepts for both architectural elements (Room, Space, Window, Stair) and structural elements (Column, Beam, Wall).
CSharpCodeExecutionService	A shared service that handles dynamic compilation and execution of AI-generated C# code using the CSharpCodeProvider class from the System.CodeDom.Compiler namespace. Manages assembly references for both Revit API and add-in types.
RuleEngineManager	Executes compliance checks using deterministic logic when matching rules with executable implementations are found in the Knowledge Graph. Shared across all compliance domains.
ElementHelper Libraries	Domain-specific helper libraries providing methods for accessing element data. The ArchitecturalElementHelper provides space and room area calculations, unit conversions, and spatial element identification. The StructuralElementHelper provides methods for accessing structural properties such as cross-section dimensions and material specifications.
RevitParameterHelper	A shared utility class that abstracts Revit parameter access, handling parameter lookup on both element instances and their types, supporting alternative parameter names, and providing value conversion utilities.

The primary orchestrator is the AiComplianceEventHandler. This module manages the logic of the system by coordinating between user input, the KG, and the LLM. It is responsible for classifying requests and invoking the appropriate services based on whether a rule already exists or needs to be generated. This module also handles the self-correction loop when generated code fails to compile. By separating the coordination of tasks from the reasoning tasks of the LLM, the framework can handle multi-step compliance workflows in a stable and reliable way.

As for the KG design, we use the Resource Description Framework (RDF) (2014a) through the dotNetRDF library (Vesse et al. 2025). All data has persisted in a single Turtle (TTL) (2014b) format file named IdasKnowledgeGraph.ttl. This supports extensibility, as new regulations and rules can be added without modifying the core system code. It also enables semantic queries that go beyond simple keyword matching.

The KG schema is built around the notion of a concept, which serves as an abstraction layer between compliance rules and the underlying BIM platform. A concept represents a domain-level understanding of a building element's functional role, independent of any specific software categorization. This abstraction is necessary because BIM authoring tools use internal categorization systems that may classify functionally identical elements into separate categories. For example, Autodesk Revit distinguishes between "Structural Columns" and "Architectural Columns" as separate categories, even though both represent vertical members. By referencing abstract domain concepts (e.g., "ColumnConcept") rather than software-specific categories, rules can apply uniformly across related element types, reducing duplication and aligning with the terminology used in building codes.

The system uses Google Gemini Pro (Gemini Team 2025) as its primary AI model, with the Claude API from Anthropic (2024) available as a fallback option to ensure resilience against service disruptions. Both experiments reported in this paper were conducted using gemini-3-pro-preview, accessed via Google's Generative Language API, in January 2026. Generation temperature was set per stage: 0.1 for rule/pattern matching (Mapping, Appendix A), 0.2 for rule-detail extraction and code generation (Generation, Appendix B), and 0.25 for the self-correction step (Debugging, Appendix C), with output-token caps of 4,096 tokens for matching and extraction and 8,192 tokens for code generation and self-correction. Decoding used top_p = 0.95; no seed is set in the request, so re-running the same request may produce slightly different outputs. To account for this, each rule was evaluated across five separate runs (Sections 5.1.2 and 5.2.2). All other decoding parameters were left at API defaults. The Claude fallback (claude-sonnet-4-5-20250929) was available but did not contribute to the reported results.

When a user submits a request in natural language, the AiComplianceEventHandler first queries the KnowledgeGraph to search for existing rules that match the request. The system examines the KG for rules containing executable implementations, specifically those with custom C# code stored via the hasCustomCSharpImplementation predicate. If a matching rule with executable logic is found, the RuleEngineManager retrieves and executes this stored implementation, bypassing the LLM at execution time. If the request cannot be matched to any rule with existing executable code, the EventHandler directs the AiCommsManager to task the LLM with generating new C# code tailored to the relevant domain.

The system implements an automated self-correction mechanism to handle compilation failures in generated code. When the CSharpCodeExecutionService attempts to compile the generated code and compilation fails, the EventHandler captures the specific compiler errors and sends them back to the LLM, requesting corrected code. This cycle repeats for up to three attempts, with attempt tracking and transparency logging to maintain auditability. The mechanism addresses only syntactic and compilation errors. The semantic correctness of the generated logic is established separately through user review of the rule's results.

Upon successful compilation, execution, and user confirmation, the EventHandler persists the new, validated logic back into the KG through the AddOrUpdateRule method. The rule is contextually integrated with relevant concepts. Subsequent requests matching the same requirement retrieve and execute the stored code directly, providing deterministic results for stored rules while allowing new rules to be generated when needed.

The architecture also supports the dynamic expansion of the KG's vocabulary. While core concepts are predefined in the graph, the CreateNewConcept method allows new concepts to be added as they are encountered during use. This enables the rule base to grow as users introduce requirements involving previously unknown element types or properties.

5. EXPERIMENTS AND RESULTS

Two experiments were conducted to examine the framework's two execution paths. Experiment 1 examines the execution of stored rules through Path A, and Experiment 2 examines the feedback loop through which new rules are generated and persisted in the KG through Path B. The experimental sequence was conducted on three models (Figure 3). Model A is a structural model sourced from the standard Revit sample project, model B is an architectural model obtained from BIMobject (2014), an open-access BIM content platform, and model C is an architectural model representing a real-world project in Israel.

In both experiments, the ground truth for each element was verified by the authors by reading its parameters and geometry directly from the BIM model, for example, element schedules, dimensions, and positions exported from Revit, and comparing them against the threshold values and conditions defined by each rule.

5.1 Experiment 1 - Path A

This experiment focuses on Path A of the system workflow (Figure 1). The goal was to demonstrate that the system can correctly retrieve a specific rule and its executable code by using the context provided in the KG. The KG formalizes each rule so that it holds both the regulation's semantic meaning and its associated executable logic, which allows the LLM to match a natural language request to the correct stored rule. A successful outcome supports the design assumption that established rules can be reused reliably without generating new code for every check.

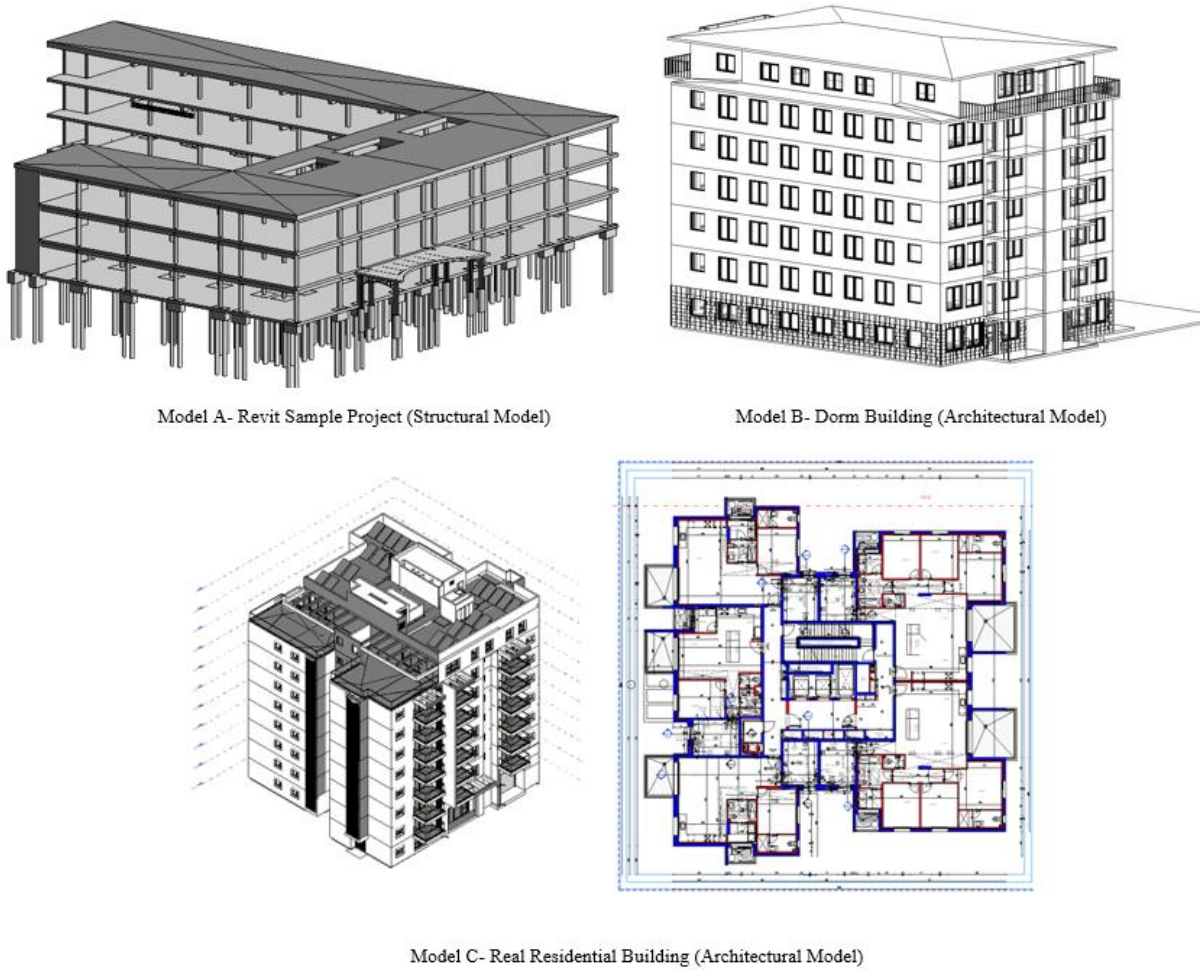

Figure 3: Revit Models used for experiments.

5.1.1 KG initialization

The KG was initialized with six compliance rules derived from three regulatory sources, representing different constraint types and building element categories. These sources include the American Concrete Institute's Building Code Requirements for Structural Concrete (ACI 318-19) (American Concrete Institute 2019), the International Building Code (IBC 2021) (International Code Council 2021), and the Israeli Home Front Command Standard for Protected Spaces (HFC 6484-197A) (Israeli Home Front Command 2010). The regulations considered are presented in Table 2.

Table 2: Regulations represented in the initial Knowledge Graph.

Rule ID	Code Reference	Threshold	Unit	Description
COLUMN_MinCoreDimension_ACI	ACI 318-19 18.7.2.1	≥ 12.0	inches	Minimum column dimension
SPACE_AREA_MIN	Israeli HFC 6484-197A	≥ 9.0	m^2	Minimum protected space area
ROOM_VOLUME_RANGE_HFC	Israeli HFC 6484-197A	22.5–36.0	m^3	Protected space volume bounds
STAIR_RISER_HEIGHT	IBC 2021 1011.5.2	4.0–7.0	inches	Riser height range
STAIR_TREAD_DEPTH	IBC 2021 1011.5.2	≥ 11.0	inches	Minimum tread depth
WINDOW_SILL_HEIGHT	IBC 2021 1015.8	≥ 36.0	inches	Minimum sill height for fall protection

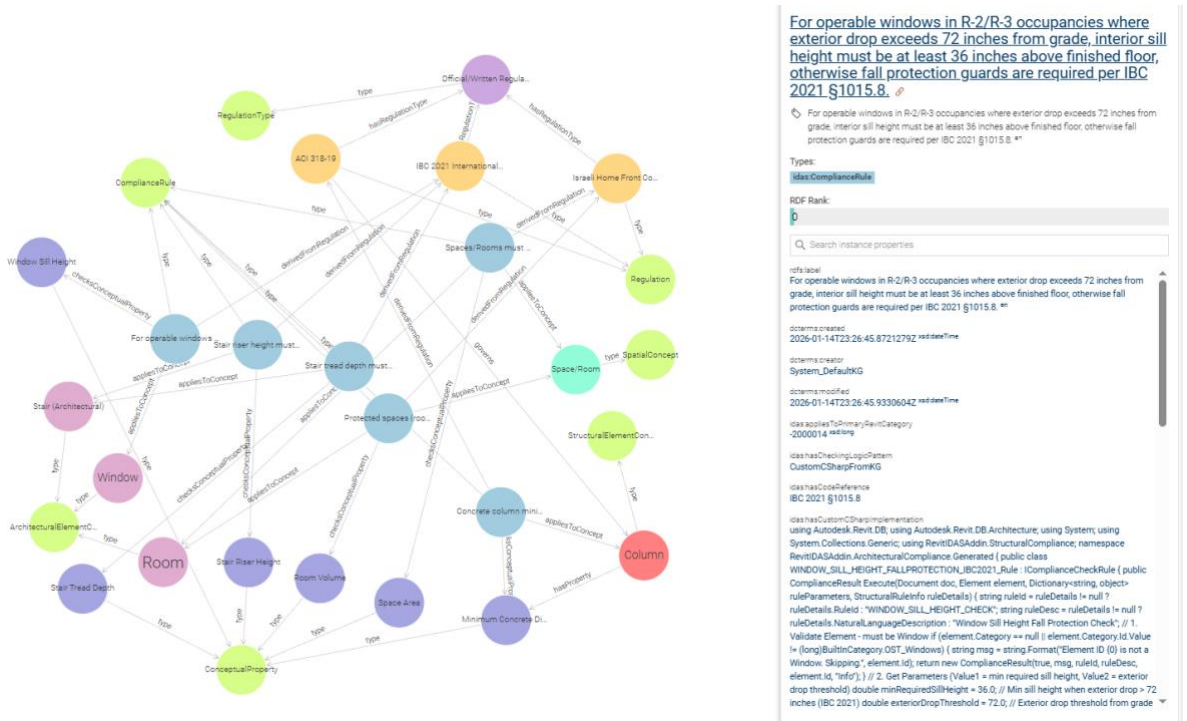

Figure 4: Initial KG populated with regulatory information and checking routines.

These rules were selected to cover different constraint types (absolute thresholds, range constraints, and conditional requirements), multiple regulatory sources, and various building element types. The initial KG is illustrated in Figure 4. To explain how the KG encodes executable compliance logic, consider the window sill height rule. As shown in Figure 4, this rule node is connected to three key entities through semantic predicates: the Window concept via appliesToConcept, the IBC 2021 regulation via derivedFromRegulation, and the WindowSillHeight property via checksConceptualProperty. Since regulation per IBC 1015.8 applies only to windows exposed to exterior conditions where the drop exceeds 72 inches above grade, this requires conditional logic. This computational logic is stored directly in the rule node via the hasCustomCSharpImplementation property, which contains the complete C# code that: (1) retrieves the window's associated rooms using Revit's FromRoom and ToRoom properties, (2) calculates the exterior drop elevation using the element's bounding box coordinates, and (3) applies the sill height threshold only when the window is exterior-facing and the drop exceeds the 72-inch threshold. When the system retrieves this rule, the stored code is compiled and executed at runtime.

5.1.2 Experiment procedure

Each rule was tested through five separate natural language requests phrased as a designer might naturally express them. The requests were deliberately varied in wording while maintaining the same semantic intent. When a request is submitted, the system executes the following sequence: (1) the EventHandler queries the KG for rules potentially relevant to the request, (2) retrieved rule summaries are packaged with the user query and sent to the LLM via the AiCommsManager, and (3) the LLM determines whether the request matches an existing rule or requires novel code generation. The mapping prompt enforces a strict two-step decision process that prioritizes rule reuse over novel generation and requires the LLM to provide chain-of-thought reasoning justifying its classification. The framework implements domain-specific variants of this prompt. Appendices A, B, and C present the architectural engineering version. An equivalent structural engineering prompt follows the same decision logic, adapted to structural design contexts. Both variants are integrated within the same tool, automatically activating the appropriate prompt based on the compliance module selected by the user.

The user interface provides two interaction modes for initiating compliance checks, as illustrated in Figure 5. In the first mode, users can select rules directly from a hierarchical tree view by checking the corresponding box. The tree view is populated using SPARQL queries executed against the in-memory RDF graph to retrieve all compliance rules with their associated metadata. When a rule is selected, the system retrieves its executable implementation directly from the KG using the dotNetRDF library's graph traversal API (GetTriplesWithSubject), which queries the in-memory RDF graph without invoking the LLM. In the second mode, users can enter requirements in natural language through the chatbot input field. This mode enables flexible querying when the exact rule name is unknown or when exploring requirements across multiple regulations. In this case, the user entered "check the windows sill height as per the IBC 2021 code" in the custom requirement field, demonstrating the natural language pathway.

Upon submitting the request, the system queries the KG, identifies a matching rule, and presents the confirmation dialog shown in Figure 6. The dialog displays the matched rule identifier (WINDOW_SILL_HEIGHT_FALLPROTECTION_IBC2021), the source regulation (IBC 2021 1015.8), and the rule's description. The Parameter Configuration section allows the user to adjust the threshold value before execution. Three options are provided: use a value extracted from the query (if specified), use the default value stored in the KG (36 inches in this case), or enter a custom value. This flexibility enables designers to apply professional judgment, for instance, checking against a more stringent company standard.

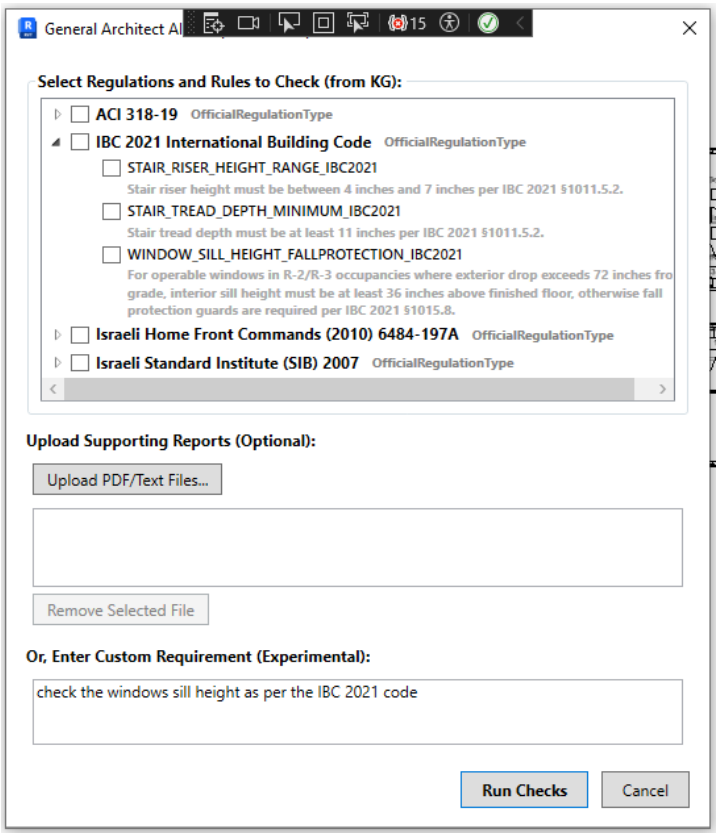

Figure 5: Experiment 1—Interaction with the system through predefined lists and natural language.

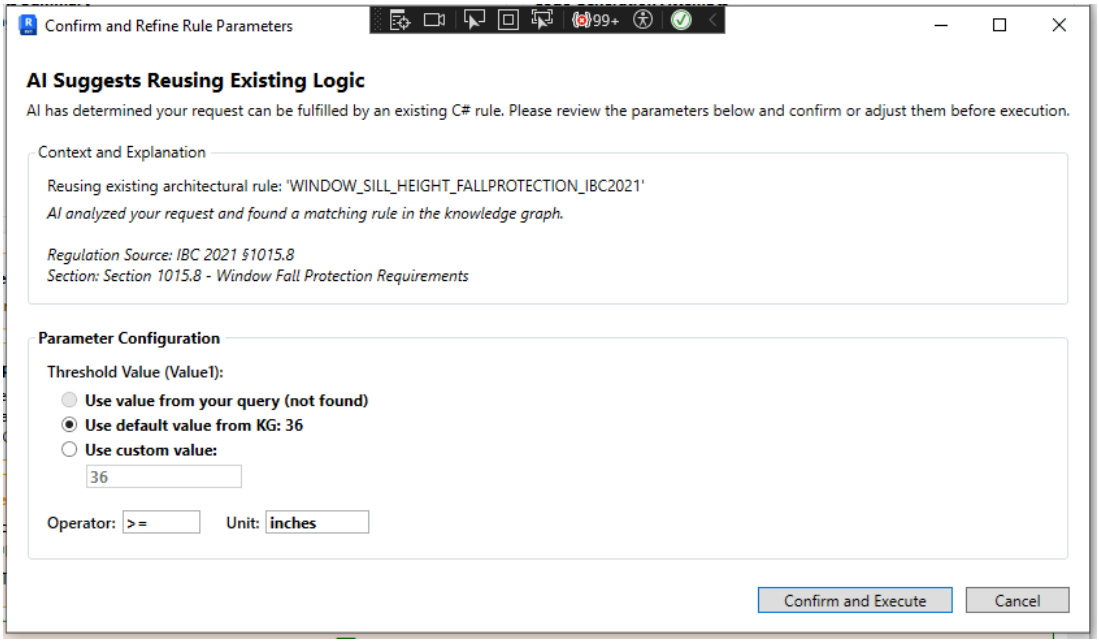

Figure 6: Experiment 1—AI reasoning and confirmation request.

5.1.3 Results

Table 3 summarizes the mapping accuracy across all six rules. The system achieved 100% accuracy in correctly identifying and retrieving the appropriate KG rule for all 30 test requests.

Table 3: Experiment 1: Rule mapping and result accuracy.

Rule #	Rule Request	Total Requests	Mapping Success	Results Accuracy (%)
1	Check the minimum dimention for the columns as per the ACI 318 Code	5	5	**100**
2	Check all security rooms have a minimum area requirement as per the Israeli Home Front Command regulations	5	5	**100**
3	Check all security rooms have a minimum volume requirement as per the Israeli Home Front Command regulations	5	5	**100**
4	Check all stair riser height as per IBC 2021 code requirements	5	5	**100**
5	Check all stair tread depth as per IBC 2021 code requirements	5	5	**100**
6	Check the windows sill height as per the IBC 2021 code	5	5	**100**

Following successful mapping, the system executed each rule against the appropriate test model (illustrated in Figure 3). The column dimension check was executed on Model A, evaluating all 203 structural columns against the ACI 318-19 minimum dimension requirement. The protected space rules (area and volume) were tested on Model C, which contains rooms designated as security spaces per Israeli civil defense requirements. The stairs, windows, and remaining architectural rules were executed on Model B.

Figure 7 presents the execution results for the window sill height check on Model B. The compliance check is based on the following logic: a window is classified as passed if (a) it provides direct access to a balcony, (b) it has an exterior drop of less than 72 inches, or (c) it has a sill height of at least 36 inches. Conversely, a window is classified as failed, indicating the need for fall protection, if the exterior drop exceeds 72 inches and the sill height is below 36 inches.

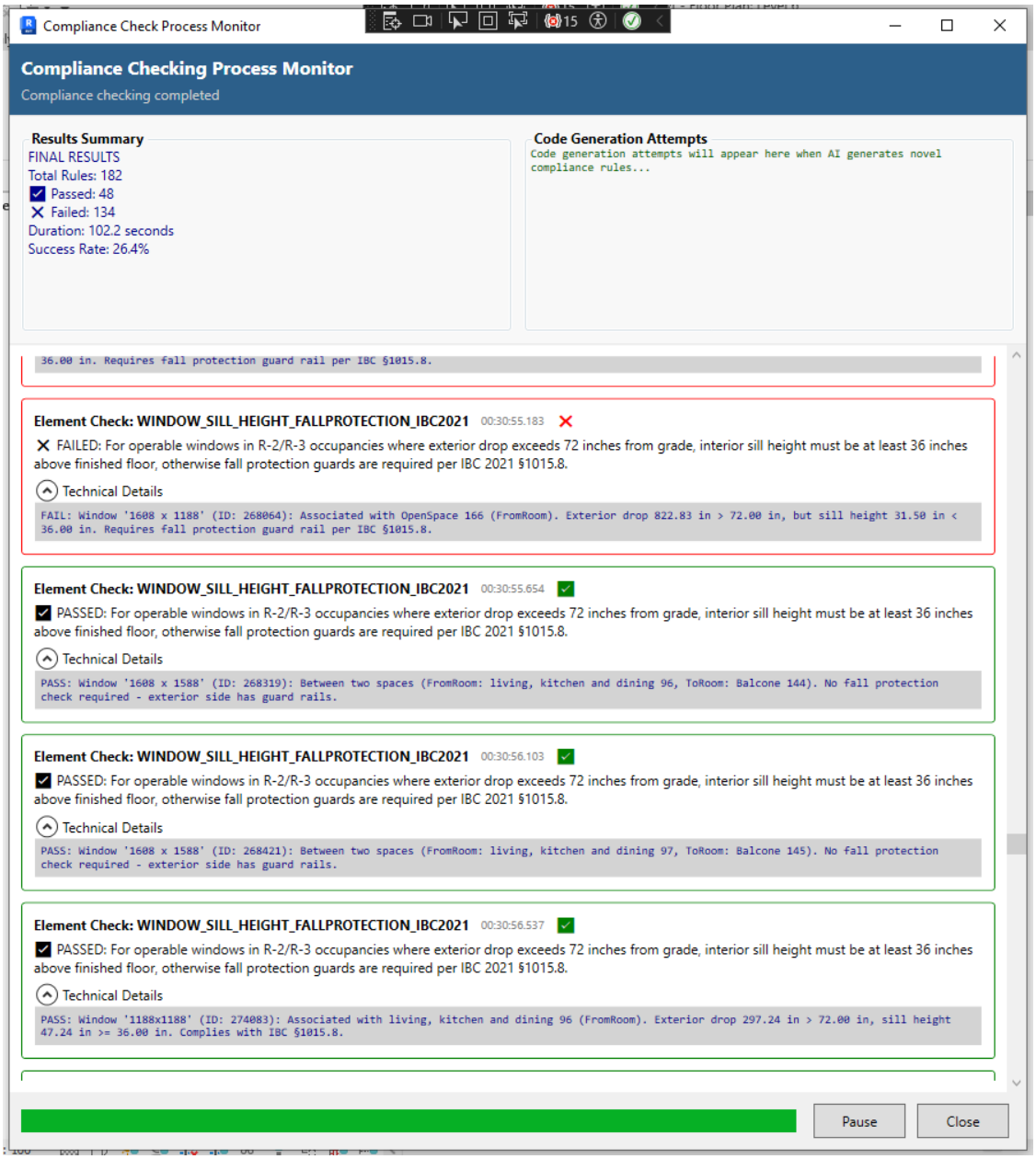

Figure 7: Experiment 1—Sample check results.

The Compliance Check Process Monitor displays a summary indicating 182 windows evaluated, with 48 passing and 134 failing the requirement. Manual validation using window schedules exported from Revit confirmed the following breakdown: 28 windows provide access to balconies and were correctly classified as passed; 20 windows are located at ground floor level with an exterior drop of less than 72 inches, correctly classified as passed; and the remaining 134 windows have an exterior drop exceeding 72 inches combined with a sill height below 36 inches, correctly classified as failed and requiring fall protection measures. This cross-verification confirmed that the system accurately applied the compliance logic to all 182 windows.

In all cases, the framework correctly filtered elements based on the rule's associated concept, retrieved the stored execution logic, and produced accurate compliance assessments. The execution results were consistent across repeated tests, with the only observed failures attributable to external LLM service disruptions rather than the framework's internal logic.

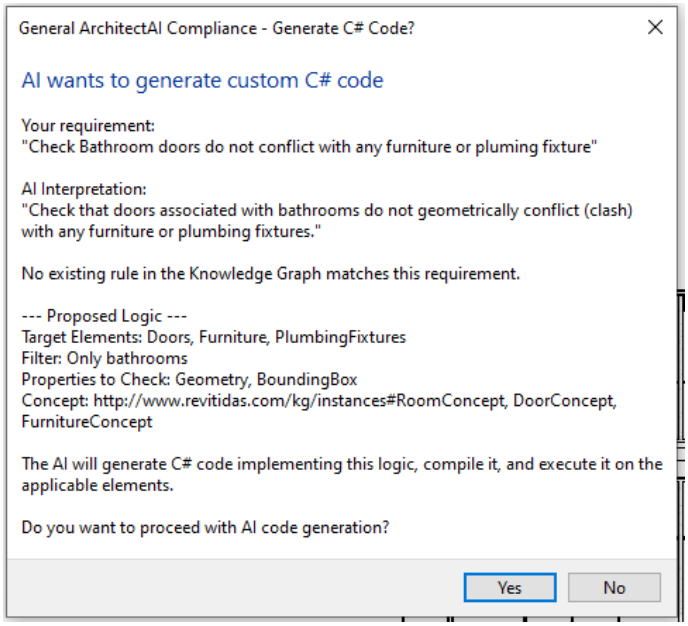

Figure 8: Experiment 2—AI reasoning for a sample request.

5.2 Experiment 2 - Path B

The second experiment focuses on Path B of the system workflow (illustrated in Figure 2), where user requests that cannot be matched to existing rules trigger code generation by the LLM. The goal is to examine the framework's ability to capture new compliance rules expressed during the design process.

5.2.1 Experiment design

Four requirements not represented in the initial KG were selected to test the system's generative capabilities across varying levels of complexity. Such requirements may originate from various sources, including project-specific constraints, organizational standards, or written regulations that have not yet been translated into executable rules. The selected requirements range from simple parameter checks to complex multi-element geometric analyses:

- Beam span-to-depth ratio: A structural engineering rule requiring calculation of derived properties and comparison against a threshold. Since "Span Ratio" is not a standard Revit parameter, custom code must retrieve the beam's length and depth, perform the division, and compare the result against the threshold.
- Door minimum width: An accessibility-related check requiring retrieval of a standard element parameter and threshold comparison.
- Bathroom door swing direction: A spatial logic rule requiring identification of bathroom spaces based on room naming conventions and verification that associated doors swing outward.
- Bathroom door clearance verification: A geometric coordination rule requiring identification of bathroom doors and detection of physical intersections with furniture or plumbing fixtures using bounding box analysis.

5.2.2 Experiment procedure

When a user submits a requirement that cannot match an existing KG rule, the system initiates the code generation workflow. The code generation workflow uses a three-stage prompt engineering strategy executed in sequence: Mapping, Generation, and Debugging.

Stage 1 - Mapping (Appendix A): The LLM is prompted to act as an analyst. The mapping prompt establishes context by assigning LLM the role of an expert assistant with a primary directive to reuse existing rules if possible. The prompt outlines mandatory steps: scan KG summaries for a rule with the same core function; if found, classify

the request for reuse by setting IsNovel to false. Only if no match exists is the request labeled as novel. The prompt requires LLM to provide chain-of-thought justification in a dedicated reasoning field. Figure 8 shows this stage for the bathroom door clearance verification rule.

Stage 2 - Generation (Appendix B): If a rule is deemed novel and the user approves, the Generation Prompt creates the initial C# code. This prompt contains the rule's requirements, the ComplianceCheckRule C# interface definition, strict coding guidelines, and reference code examples retrieved directly from the KG. These guidelines instruct the LLM on correct Revit API access patterns and a small set of syntax error avoidance that were observed during early implementation work.

Stage 3 - Debugging (Appendix C): The CSharpCodeExecutionService attempts to compile the generated code. If compilation fails, the system captures the specific compiler errors and initiates an automated self-correction loop. The debugging prompt provides the LLM with the original failed code and the exact compiler error messages, with instructions shifting from generation to correction. Figure 9 demonstrates this process.

5.2.3 Results

Table 4 presents detailed results across all four rules and 20 total generation attempts. The metrics capture target element recognition, compilation success at each attempt, result accuracy upon successful compilation, and final outcome. A request is considered successful only when the generated code both compiles and produces correct compliance assessments. All rules were manually checked to provide ground truth for system performance evaluation. Table 5 provides a summary of the attempt success rates.

The results show a correlation between rule complexity and generation success. Simple parameter-based checks achieved 100% success rates (Table 5). The beam span-to-depth ratio rule, despite requiring derived property calculation, benefited from familiar structural engineering semantics and standard Revit API patterns for accessing beam dimensions. All five attempts succeeded on the first try, with manual verification confirming the results.

The door minimum width checks similarly achieved 100% success across all five requests. One request required the debugging loop due to initial syntax errors, illustrating the self-correction mechanism in operation. Upon successful compilation, the system evaluated all 165 doors in the test model, identifying 23 non-compliant elements.

Table 4: Experiment 2: Code Generation Results.

Rule Request	Request No.	Recognized Target Elements	Attempt 1 – Success	Attempt 2 – Success	Attempt 3 – Success	Final Outcome	Comments
Check Concrete beams have span ratio less than 20	1	✓	✓	–	–	Success	
	2	✓	✓	–	–	Success	
	3	✓	✓	–	–	Success	
	4	✓	✓	–	–	Success	
	5	✓	✓	–	–	Success	
Check all the doors have minimum width of 32 inches	1	✓	✓	–	–	Success	
	2	✓	✓	–	–	Success	
	3	✓	✓	–	–	Success	

	4	✓	✗	✓	–	Success	
	5	✓	✓	–	–	Success	
check all doors direction for the bathrooms are open outside	1	✓	✗	✗	✗	Failed	Exceeded attempts
	2	✓	✓	-	-	Success	
	3	✓	✓	–	–	Success	
	4	✓	✓	–	–	Success	
	5	✓	✓	–	–	Failed	Incorrect compliance results
Check Bathroom doors do not conflict with any furniture or pluming fixture	1	✓	✗	✓	–	Success	
	2	✓	✗	✗	✓	Failed	Incorrect compliance results
	3	✓	✓	-	-	Failed	Incorrect compliance results
	4	✓	✓	-	-	Failed	Incorrect compliance results
	5	✓	✗	✗	✓	Failed	Incorrect compliance results

Table 5: Experiment 2: Summary of results.

Rule #	**Rule Request**	**Total Requests**	**Successful Code generation and results**	**Rate of Success (%)**
1	Beam span-to-depth ratio	5	5	**100**
2	Door minimum width	5	5	**100**
3	Bathroom door swing direction	5	3	**60**
4	Bathroom door clearance verification	5	1	**20**
	Overall	20	14	**70**

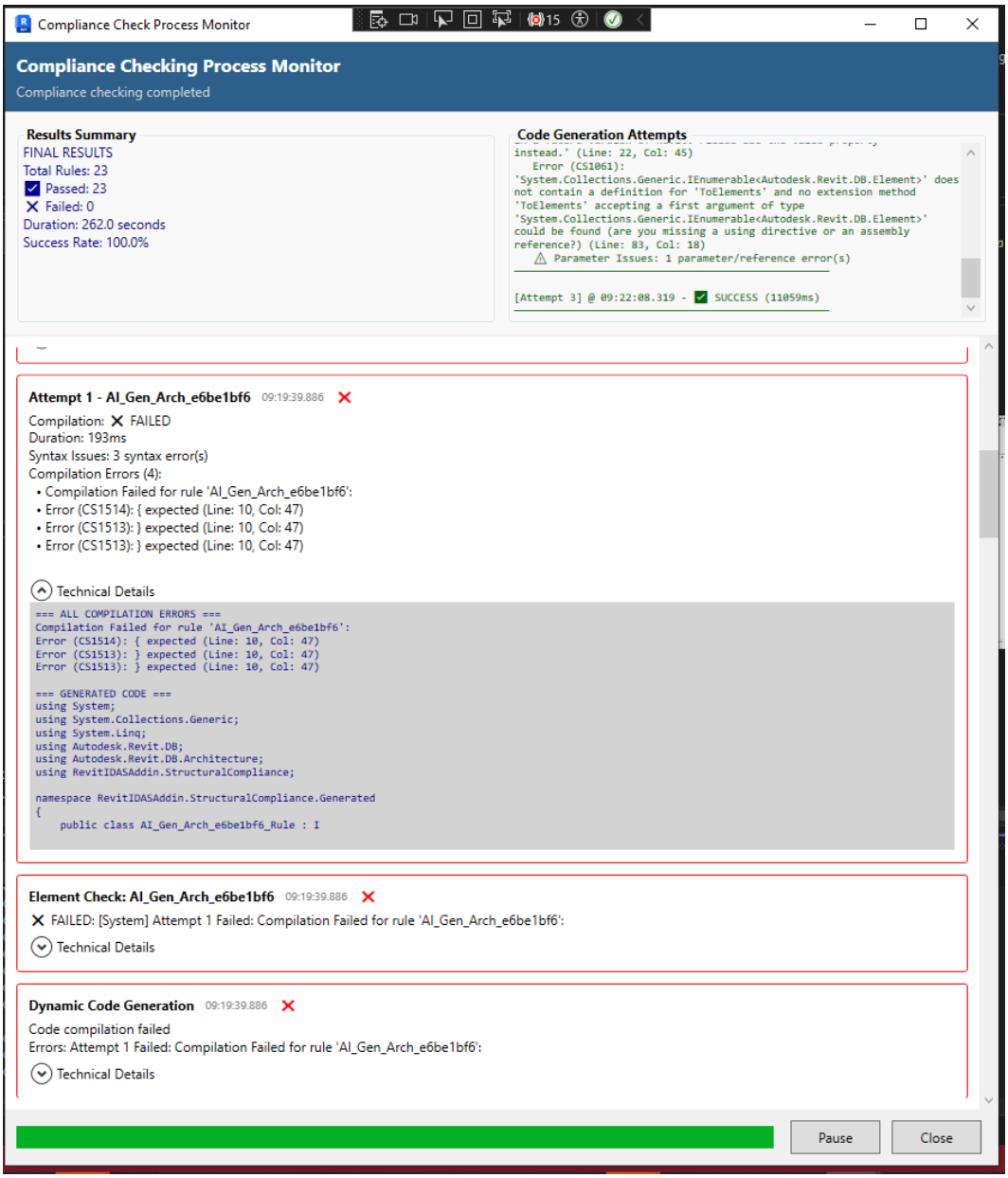

Figure 9: Debug errors (Sample).

More complex rules involving spatial reasoning showed decreased reliability. The bathroom door swing direction check achieved 60% success (3 of 5 requests). In the test model, all bathroom doors were correctly designed to open outward, meaning a correct implementation should report 100% compliance. Three requests successfully generated code that correctly identified all bathroom doors as compliant. One request failed due to compilation errors exceeding the three-attempt limit. In a separate request, the code compiled successfully and executed without errors but produced incorrect results, flagging compliant doors as violations. The latter case illustrates a failure mode in which compilation success does not guarantee semantic correctness, and points to the role of human validation in the workflow. This failure mode might suggest that the system could be improved by using specialized agents designed to handle the semantic alignment between the generated code and the actual building model.

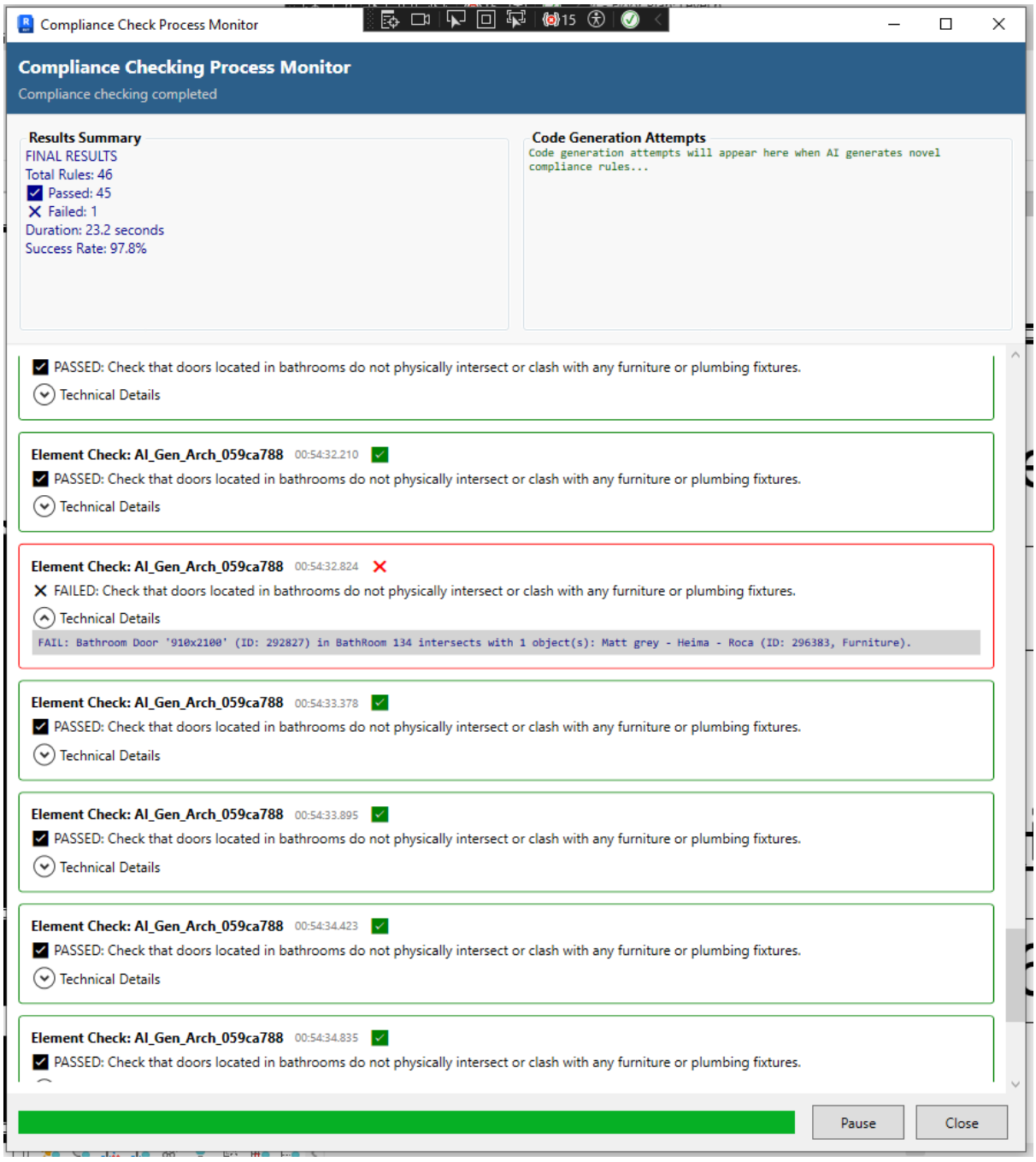

Figure 10: Experiment 2-Sample check results.

The bathroom door clearance verification rule, requiring geometric intersection analysis across multiple element types, achieved 20% success (1 of 5 requests). Figure 10 shows the single successful execution: 46 bathroom doors and fixtures evaluated, with 45 passing and 1 failing. The remaining four attempts produced incorrect results despite successful compilation, indicating that the LLM's difficulty lay not in code generation itself but in reasoning over the geometry, topology, and semantics of the BIM model.

Target element recognition achieved 100% accuracy across all 20 attempts, confirming that the mapping prompt reliably interprets user intent even for complex multi-element requirements. The failures occurred downstream in the code generation stage, not in requirement understanding.

Figure 11 shows the expanded KG after incorporating the successfully generated rules. The graph now organizes compliance rules along two dimensions. First, rules are categorized by domain, structural and architectural. Second, rules are distinguished by origin: official rules are derived from established building codes, while custom rules are generated by LLM in response to designer requirements. This classification is preserved to ensure quality and traceability of the resulting KG. Official rules reference their source regulation via the derivedFromRegulation predicate, while LLM-generated rules are marked with dcterms:creator set to AI_UserApproved and include timestamps recording their creation date. The domain context is maintained through the system's prompt architecture, with separate structural and architectural compliance modules activating the appropriate domain-specific prompts as described in Experiment 1. All rule types share the same structural integration, with each node linked to applicable concepts and storing executable C# code in the hasCustomCSharpImplementation property. This unified architecture enables both official standards and LLM-generated rules to become regulatory assets, available for reuse through the Path A workflow. The graph thus evolves from a static repository of codified regulations into a dynamic, multi-domain knowledge base that captures official standards alongside organization-specific design requirements expressed by practitioners.

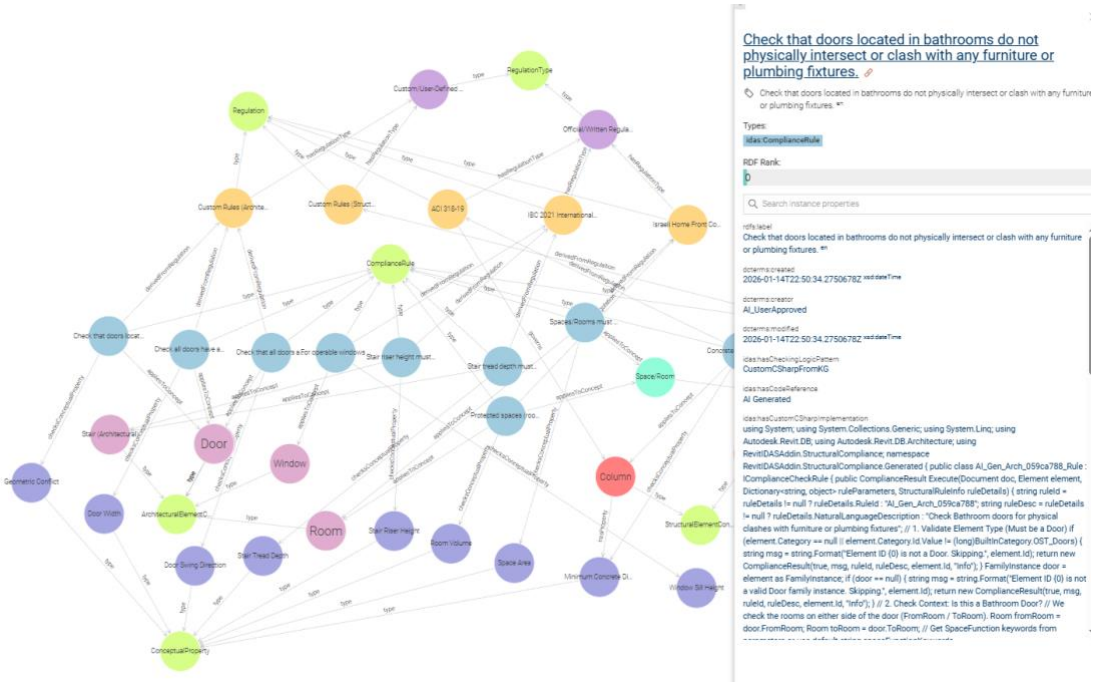

Figure 11: Experiment 2-Expanded Knowledge Graph.

To confirm this, each successfully generated rule was subsequently retrieved and executed through Path A workflow on a different model from the one on which it was generated. In every case, the stored logic was recalled and executed deterministically, without re-invoking the LLM for code generation, and produced results matching manual verification, confirming that validated Path B rules become reusable Path A assets.

6. DISCUSSION

This work aims to support a design review process that can be extended with new validation rules during use, rather than being limited to a set of rules defined before deployment. We proposed and tested a framework that moves compliance checking from a static, post-design audit towards a flexible, continuous feedback system integrated into the design environment. The central idea is to allow the rule base to grow as new requirements emerge and let designers express what they need without requiring them to write code.

Current research has primarily focused on LLMs as tools for automating the digitization of formal regulatory text. For example, some approaches generate code to update non-compliant values against a pre-constructed KG (Feng et al. 2025), maintain a SHACL-based rule base through semantic extraction (Zhang et al. 2026), or utilize the LLM as a core reasoning engine supported by a library of pre-written extraction scripts (Iversen and Huang 2026).

One limitation of these prior efforts is that the designer has no direct way to add or modify rules. The rule base grows only through developer-managed pipelines. The framework presented in this work proposes an interactive, agentic channel for expanding the rule set. When a designer encounters a project-specific constraint or a requirement not yet captured by the system, they express it in natural language. The system generates executable code for immediate validation. Once the designer confirms the result, the logic persisted as an asset within the KG for future use. Consequently, the rule base becomes a dynamic resource that evolves through designer experience, allowing rules to be reused deterministically in future design cycles without re-invoking the LLM for code generation. This reuse was verified experimentally: generated rules were re-executed through Path A on separate models and produced consistent, deterministic results. One consequence of letting designers author rules during their own work is that the framework can accept rules that do not exist in any formal document. Company standards, project-specific requirements, and design rules of thumb adopted from professional practice are routinely applied during design, but they are rarely written down in a form that an ACC system can ingest. Existing systems have no mechanism to accept such rules during design, which is when they typically arise. In the framework presented here, a designer who wants to check one of these rules states it in natural language, and once the rule is validated, it becomes available to other designers on later projects under the same mechanism as a rule derived from a formal code.

Beyond accepting new rules, the framework also lets designers adjust the threshold values of existing ones. As shown earlier in Figure 6, the threshold attached to a stored rule can be set before the rule is run, so the same rule can be applied with the value held in the rule base, a stricter company standard, or a value specific to the project at hand, without modifying the underlying rule logic. The same mechanism supports adaptation when a regulatory update changes only a threshold value, since the rule logic remains the same.

This can be considered as both an advantage and a risk, since unconstrained flexibility could compromise the integrity of the knowledge base. To mitigate this risk, each time an expert approves a new AI-generated rule, they are confirming that the logic is correct and valuable, ensuring the knowledge base evolves accurately under human supervision. It is important to note that the validation does not necessarily require programming expertise. A domain expert can verify logic correctness by examining compliance results against known conditions in the model. An effective strategy is to intentionally create violations in a test model and confirm that the rule correctly identifies them. Ultimately, the framework's reliability depends strongly on its responsible use. Without thoughtful oversight and professional judgment, the system cannot guarantee the validity of its accumulated knowledge.

A critical drawback of the system is the probabilistic nature of LLMs (He et al. 2025; Joffe et al. 2025). Advanced prompt engineering is required to achieve consistent and reliable mapping behavior. In this work, we refined the mapping prompt in the AiCommsManager to enforce a strict, step-by-step decision process. To improve reliability, we mandated a "chain-of-thought" approach by requiring the LLM to include a reasoning field in its JSON output, compelling it to justify its decision-making process.

A similar strategy was implemented for the code generation workflow. The initial outputs from the LLM were often syntactically flawed, a common issue in AI-based code generation, and an acknowledged challenge in related work (Madireddy et al. 2025). In the proposed framework, compilation errors are not treated as terminal failures. Instead, the DebugCSharpCode method initiates a self-correction loop in which compiler errors are captured and supplied as context for a new prompt that requests corrected code. This loop addresses syntactic and compilation errors only; semantic correctness, whether the code actually performs the intended check, is not addressed by the loop itself and is instead established through the user validation step described above. Evidently, there is a trade-off in prompt design between grounding the LLM to improve reliability (He et al. 2025; Joffe et al. 2025). Excessive constraints can negatively affect reasoning performance. For instance, when the mapping prompt was made overly restrictive, we observed instances where the LLM failed to identify a clear functional match to an existing KG rule, defaulting instead to an incorrect "novel rule" declaration. Consequently, our prompting approach was to strongly guide the model without completely eliminating its reasoning flexibility.

Despite this tuning, our results indicate that achieving a 100% success rate remains a challenge. As described in the results section, the system achieved reliable performance for well-defined parameter-based rules and successfully recovered from compilation errors through the self-correction loop. However, complex geometric and spatial reasoning rules exhibited lower success rates, with the primary failure mode being semantically incorrect logic rather than compilation errors. These findings point to a deeper limitation that cannot be addressed through prompt engineering or fine-tuning alone. LLMs are fundamentally designed for language processing, not for

reasoning over geometry, topology, or structured building semantics. Consequently, when applied directly to BIM models, they struggle to construct correct spatial and relational interpretations, leading to semantically flawed reasoning even when the generated logic is syntactically valid.

The proposed framework does not blindly trust the LLM's output. Results of our experiments show that while compilation errors can be detected and corrected automatically by the system, identifying semantically incorrect code requires the input of the user. The designer must therefore review and verify execution results before approving new rules for permanent storage in the KG, serving as a critical safeguard against propagating flawed logic into the reusable knowledge assets. This model of AI-human collaboration offers a practical pathway for safely integrating probabilistic models into engineering practice.

6.1 Limitations and future work

The central limitation of this framework lies in the ability of the LLM to handle requirements involving geometric reasoning and spatial relationships. As demonstrated in the experiments, the system achieved 100 percent success for parameter-based checks, but reliability decreased significantly for complex requirements (60 percent and 20 percent, respectively. For the bathroom door clearance rule, the LLM consistently emitted syntactically valid C# logic, yet the executions returned semantically incorrect results. This confirms that the bottleneck is not in code generation, but in the reasoning required to understand how building elements occupy space and interact with one another.

Complex compliance often involves the aggregation of multiple element types and interconnected conditions that span different building systems, which were not explored in this work. To address this, the current agentic structure could be extended with more specialized components, some of them more suited for handling geometric and spatial reasoning, while the LLM continues to focus on the dialogue with the designer. This structure would potentially allow the system to query the necessary geometric and topological context rather than tasking the LLM with performing spatial analysis directly.

To achieve this, the system would require richer contextual information about model geometry and element relationships. Advanced prompt engineering appears insufficient to bridge this gap, as the necessary relational context is often absent from textual model descriptions. A potential solution involves Graph RAG architectures (Iranmanesh et al. 2025), which demonstrated significant promise for querying complex building model data, such as IFC files. This would require developing a system capable of translating a user's multi-element request into a formal query to retrieve the specific set of related components (e.g., only walls that bound a specific room) from a semantic model of the project. Du et al. (2026), in their Text2BIM multi-agent framework for BIM model generation, observed that LLMs struggle with spatial understanding of the overall building, producing hallucinations when accurate placement requires comprehending the building's wider spatial layout, and propose graph-based representations of the BIM model as a direction for helping the LLM understand the spatial and topological relationships within it. Building on this direction, we assume that representing the BIM model as a Labeled Property Graph (LPG) (Zhu et al. 2022) could provide the LLM with richer contextual understanding. In an LPG representation, building elements become nodes with properties, and their spatial, functional, and hierarchical relationships become typed edges. This graph structure could be queried to extract relevant subgraphs showing element relationships, adjacencies between spaces, or host-hosted relationships. Providing such structured relational context could significantly improve the LLM's ability to generate correct logic for complex spatial requirements. This retrieved, structured data would then be used in an advanced multi-prompt chain designed to guide the LLM's reasoning. Therefore, future work will focus on broadening the framework's application scope to deal with more realistic scenarios with the support of an LPG. An LPG representation aligned with the IFC schema is being explored as part of this future direction, which would also reduce the framework's current dependency on the Revit API and broaden its applicability to other BIM authoring environments.

The system's input channels could also be extended. As illustrated in Figure 5, the user interface includes a section for uploading supporting documents. This points to a planned extension in which a designer could upload a project-specific report (e.g., as a PDF) and have the system infer the relevant constraints from the document using RAG. Such an extension would shift the framework from capturing explicitly stated intent to interpreting documented requirements that have not been individually translated into rules.

7. CONCLUSIONS

This work presents a framework for design review in which compliance rules can be introduced during the design process, executed against a BIM model, and retained as reusable assets. The framework addresses a gap in current ACC tools, which are confined to predefined rule libraries and lack a mechanism for incorporating new requirements during design. By integrating KGs and LLMs, the framework supports rules drawn from formal building codes, project-specific requirements, organizational standards, and other natural-language sources, and stores each validated rule together with its executable checking logic for deterministic reuse on subsequent projects.

The framework provides an interactive pathway for compliance checking. It enables the capture of new requirements expressed in natural language, the generation and self-correction of the corresponding C# checking logic and persist the validated rule in the KG for subsequent reuse. The framework accepts rules regardless of whether they originate from formal regulations, organizational standards, or design considerations expressed by practitioners during a project. This allows the rule base to evolve in response to needs that arise during design, rather than remaining a static repository maintained solely through offline developer updates.

A prototype add-in for Autodesk Revit was implemented to evaluate the framework through two experiments. Experiment 1 achieved 100% mapping accuracy across paraphrased natural-language requests for the six initial rules in the KG, confirming deterministic execution of stored rules through Path A. Experiment 2 evaluated novel-rule generation through Path B: simple parameter-based checks achieved 100% success, while rules involving geometric and spatial reasoning showed lower success rates (20–60%), reflecting the current limitations of LLMs in reasoning about geometric context from textual descriptions alone. Rules that were successfully generated and stored were further confirmed to be reusable through the Path A workflow on separate models, closing the generate–validate–reuse loop that the framework is built around.

Future work will focus on improving the handling of geometric and spatial reasoning by transitioning toward an agentic architecture supported by Graph RAG techniques and LPG representations. These methods aim to provide richer geometrical and relational context to the LLM, with initial efforts to align the LPG with the IFC schema currently underway. Additional directions include extending the framework to incorporate document analysis for automatic constraint extraction, broadening the evaluation through larger rule sets and user studies with practicing designers, and porting the architecture to platforms based on open standards. Together, these directions aim to mature the framework from a proof-of-concept prototype toward a system that can support compliance review across the scale and variety of real architectural and structural design practice.

DATA AVAILABILITY

The prototype source code, including the LLM prompts, and the knowledge-graph Turtle file generated and used in this study form part of a system under ongoing development and are available from the corresponding author upon request.

DECLARATION OF GENERATIVE AI AND AI-ASSISTED TECHNOLOGIES IN THE MANUSCRIPT PREPARATION PROCESS

During the preparation of this work the authors used Google's AI tools in order to enhance the language and clarity of the manuscript. The author also used Anthropic's Claude in order to assist in debugging and correcting syntactical errors in the developed code. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the published article.

REFERENCES

ACCORD Project. 2025. “ACCORD.” Accessed November 1, 2025. https://accordproject.eu/.

Alnuzha, M., and T. Bloch. 2025. “The role of machine learning in automated code checking – a systematic literature review.” Journal of Information Technology in Construction, 30: 22–44. https://doi.org/10.36680/j.itcon.2025.002.

American Concrete Institute. 2019. 318-19 Building Code Requirements for Structural Concrete and Commentary. American Concrete Institute.

Amor, R., and J. Dimyadi. 2021. “The promise of automated compliance checking.” Developments in the Built Environment, 5: 100039. https://doi.org/10.1016/j.dibe.2020.100039.

Anthropic, C. 2024. “Introducing the next generation of Claude.” Accessed January 17, 2026. https://www.anthropic.com/news/claude-3-family.

Autodesk. 2024. “Revit.” Accessed November 1, 2025. https://help.autodesk.com/view/RVT/2024/ENU/.

Beach, T. H., J.-L. Hippolyte, and Y. Rezgui. 2020. “Towards the adoption of automated regulatory compliance checking in the built environment.” Automation in Construction, 118: 103285. https://doi.org/10.1016/j.autcon.2020.103285.

BIMobject. 2014. “BIM objects - Free download! Apartment building Revit 2014 | BIMobject.” BIMobject®. Accessed February 1, 2026. https://www.bimobject.com/en-us/bimobject-models/product/revit-apartment-building?software=revit.

Bloch, T., M. Katz, R. Yosef, and R. Sacks. 2019. “Automated model checking for topologically complex code requirements – security room case study.” In: Proceedings of the 2019 European Conference on Computing in Construction, Proceedings of the European Conference on Computing in Construction, edited by J. O’Donnell, A. Chassiakos, D. Rovas, and D. Hall, 48–55.

Bloch, T., and R. Sacks. 2018. “Comparing machine learning and rule-based inferencing for semantic enrichment of BIM models.” Automation in Construction, 91: 256–272. Elsevier BV. https://doi.org/10.1016/j.autcon.2018.03.018.

Brown, T., B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, and others. 2020. “Language models are few-shot learners.” Advances in neural information processing systems, 33: 1877–1901.

Chen, G., and Y. Luo. 2016. “A BIM and ontology-based intelligent application framework.” In: 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), 494–497. Xi’an, China: IEEE.

Chen, J., and Y. Bao. 2025. “Multi-agent large language model framework for code-compliant automated design of reinforced concrete structures.” Automation in Construction, 177: 106331. https://doi.org/10.1016/j.autcon.2025.106331.

Dimyadi, J., W. Solihin, C. Eastman, and R. Amor. 2016. “Integrating the BIM Rule Language into Compliant Design Audit Processes.” In: InProceedings of the 33th CIB W78 international conference 2016 Oct 31, (pp. 1-10).

Dinis, F. M., J. Poças Martins, A. S. Guimarães, and B. Rangel. 2022. “BIM and Semantic Enrichment Methods and Applications: A Review of Recent Developments.” Archives of Computational Methods in Engineering, 29 (2): 879–895. Springer Science and Business Media LLC. https://doi.org/10.1007/s11831-021-09595-6.

Du, C., S. Esser, S. Nousias, and A. Borrmann. 2026. “Text2BIM: Generating Building Models Using a Large Language Model-Based Multiagent Framework.” Journal of Computing in Civil Engineering, 40 (2): 04025142. https://doi.org/10.1061/JCCEE5.CPENG-6386.

Eastman, C. 1975. “The Use of Computers Instead of Drawings in Building Design.” AIA Journal, 63: 46–50.

Eastman, C., J. Lee, Y. Jeong, and J. Lee. 2009. “Automatic rule-based checking of building designs.” Automation in Construction, 18 (8): 1011–1033. Elsevier BV. https://doi.org/10.1016/j.autcon.2009.07.002.

Edwards, N., J. Chauvin, and R. Blanchet. 2019. “Advocating for improvements to building codes for the population’s health.” Canadian Journal of Public Health, 110 (4): 516–519. https://doi.org/10.17269/s41997-019-00191-7.

Feng, T., X. Xu, Y. Zhang, Z. He, W. Li, and P. F. Yuan. 2025. “From Detection to Update: A framework combining LLMs and knowledge graph for automated compliance in BIM model.” 111–120. Tokyo.

Fitkau, I., and T. Hartmann. 2024. “An ontology-based approach of automatic compliance checking for structural fire safety requirements.” Advanced Engineering Informatics, 59: 102314. Elsevier BV. https://doi.org/10.1016/j.aei.2023.102314.

Gao, L., A. Madaan, S. Zhou, U. Alon, P. Liu, Y. Yang, J. Callan, and G. Neubig. 2023. “PAL: program-aided language models.” In: Proceedings of the 40th International Conference on Machine Learning, ICML’23. Honolulu, Hawaii, USA: JMLR.org.

Gemini Team. 2025. “Gemini: A Family of Highly Capable Multimodal Model.” arXiv:2312.11805v5.

Guo, T., X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang. 2024. “Large Language Model based Multi-Agents: A Survey of Progress and Challenges.” arXiv.

He, C., W. He, M. Liu, S. Leng, and S. Wei. 2025. “Enriched Construction Regulation Inquiry Responses: A Hybrid Search Approach for Large Language Models.” Journal of Management in Engineering, 41 (3). American Society of Civil Engineers (ASCE). https://doi.org/10.1061/jmenea.meeng-6444.

Hettiarachchi, H., A. Dridi, M. M. Gaber, P. Parsafard, N. Bocaneala, K. Breitenfelder, G. Costa, M. Hedblom, M. Juganaru-Mathieu, T. Mecharnia, S. Park, H. Tan, A.-R. H. Tawil, and E. Vakaj. 2025. “CODE-ACCORD: A Corpus of building regulatory data for rule generation towards automatic compliance checking.” Scientific Data, 12 (1): 170. https://doi.org/10.1038/s41597-024-04320-x.

Hevner, {Alan R.}, {Salvatore T.} March, J. Park, and S. Ram. 2004. “Design science in information systems research.” MIS Quarterly: Management Information Systems, 28 (1): 75–105. Management Information Systems Research Center. https://doi.org/10.2307/25148625.

Hjelseth, E., and N. Nisbet. 2011. “Capturing normative constraints by use of the semantic mark-up RASE methodology.” In: InProceedings of CIB W78-W102 Conference 2011 Oct, 1–10.

International Code Council. 2021. “2021 International Building Code (IBC).” Accessed January 19, 2026. https://codes.iccsafe.org/content/IBC2021V2.0.

Iranmanesh, S., H. Saadany, and E. Vakaj. 2025. “LLM-assisted Graph-RAG Information Extraction from IFC Data.” In: 2025 European Conference on Computing in Construction. Porto, Portugal.

Isaev, M., N. McDonald, and R. Vuduc. 2023. “Scaling Infrastructure to Support Multi-Trillion Parameter LLM Training.” In Architecture and System Support for Transformer Models (ASSYST@ ISCA 2023).

Ismail, A. S., K. N. Ali, N. A. Iahad, M. A. Kassem, and N. T. Al-Ashwal. 2023. “BIM-Based Automated Code Compliance Checking System in Malaysian Fire Safety Regulations: A User-Friendly Approach.” Buildings, 13 (6): 1404. MDPI AG. https://doi.org/10.3390/buildings13061404.

Israeli Home Front Command. 2010. Israeli Home Front Command Regulations. Israel: Home Front Command.

Iversen, O., and L. Huang. 2026. “Leveraging large language models for BIM-based automated compliance checking.” Automation in Construction, 182: 106707. https://doi.org/10.1016/j.autcon.2025.106707.

Izbash, Y., and V. Babayev. 2024. “Digital Evolution in AEC industry: Bridging BIM, Building Codes, and Future Technologies.” IOP Conference Series: Earth and Environmental Science, 1376 (1): 012004. IOP Publishing. https://doi.org/10.1088/1755-1315/1376/1/012004.

Joffe, I., G. Felobes, Y. Elgouhari, M. Talebi Kalaleh, Q. Mei, and Y. H. Chui. 2025. “The Framework and Implementation of Using Large Language Models to Answer Questions about Building Codes and Standards.” Journal of Computing in Civil Engineering, 39 (4). American Society of Civil Engineers (ASCE). https://doi.org/10.1061/jccee5.cpeng-6037.

Khemlani, L. 2005. “CORENET e-PlanCheck: Singapore’s Automated Code Checking System-- AECbytes Article.” AECbytes. Accessed November 2, 2025. https://www.aecbytes.com/feature/2005/CORENETePlanCheck.html.

Lee, Y.-C., P. Ghannad, J. Dimyadi, J.-K. Lee, W. Solihin, and J. Zhang. 2020. “A Comparative Analysis of Five Rule-Based Model Checking Platforms.” In: Construction Research Congress 2020, 1127–1136. Tempe, Arizona: American Society of Civil Engineers.

Lewis, P., E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela. 2020. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks."

Li, H., R. Yang, S. Xu, Y. Xiao, and H. Zhao. 2024. "Intelligent Checking Method for Construction Schemes via Fusion of Knowledge Graph and Large Language Models." Buildings, 14 (8): 2502. MDPI AG. https://doi.org/10.3390/buildings14082502.

Li, J., and A. Maiti. 2025. "Applying Large Language Model Analysis and Backend Web Services in Regulatory Technologies for Continuous Compliance Checks." Future Internet, 17 (3): 100. MDPI AG. https://doi.org/10.3390/fi17030100.

Ma, Z., H. Zhu, X. Xiang, Ž. Turk, and R. Klinc. 2024. "Automatic compliance checking of BIM models against quality standards based on ontology technology." Automation in Construction, 166: 105656. Elsevier BV. https://doi.org/10.1016/j.autcon.2024.105656.

Madireddy, S., L. Gao, Z. U. Din, K. Kim, A. Senouci, Z. Han, and Y. Zhang. 2025. "Large Language Model-Driven Code Compliance Checking in Building Information Modeling." Electronics, 14 (11): 2146. MDPI AG. https://doi.org/10.3390/electronics14112146.

Minaee, S., T. Mikolov, N. Nikzad, M. Chenaghlu, R. Socher, X. Amatriain, and J. Gao. 2024. "Large Language Models: A Survey." arXiv e-prints, arXiv:2402.06196. https://doi.org/10.48550/arXiv.2402.06196.

Mu, F., L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang. 2024. "ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification." Proceedings of the ACM on Software Engineering, 1 (FSE): 2332–2354. https://doi.org/10.1145/3660810.

Nakhaee, A., D. Elshani, and T. Wortmann. 2024. "A Vision for Automated Building Code Compliance Checking by Unifying Hybrid Knowledge Graphs and Large Language Models." Scalable Disruptors, 445–457. Cham: Springer Nature Switzerland.

Pauwels, P., S. Zhang, and Y.-C. Lee. 2017. "Semantic web technologies in AEC industry: A literature overview." Automation in Construction, 73: 145–165. Elsevier BV. https://doi.org/10.1016/j.autcon.2016.10.003.

Peng, J., and X. Liu. 2023. "Automated code compliance checking research based on BIM and knowledge graph." Scientific Reports, 13 (1). Springer Science and Business Media LLC. https://doi.org/10.1038/s41598-023-34342-1.

Preidel, C., and A. Borrmann. 2015. "Automated Code Compliance Checking Based on a Visual Language and Building Information Modeling." In: 32nd International Symposium on Automation and Robotics in Construction. Oulu, Finland.

Preidel, C., and A. Borrmann. 2016. "Integrating Relational Algebra into a Visual Code Checking Language for Information Retrieval from Building Information Models." Unpublished. https://doi.org/10.13140/RG.2.1.4618.5201.

Preidel, C., and A. Borrmann. 2018. "BIM-Based Code Compliance Checking." Building Information Modeling, edited by A. Borrmann, M. König, C. Koch, and J. Beetz, 367–381. Cham: Springer International Publishing.

Sacks, R., M. Girolami, and I. Brilakis. 2020. "Building Information Modelling, Artificial Intelligence and Construction Tech." Developments in the Built Environment, 4: 100011. Elsevier BV. https://doi.org/10.1016/j.dibe.2020.100011.

Solibri. 2025. "Solibri | BIM software for architects, engineers and construction." Accessed November 1, 2025. https://www.solibri.com.

Tang, C., K.-M. Tam, and C. Herr. 2025. "Archi-Agents Approximating the architectural design process with collaborative specialized multi- agent system llm and building information modeling." In: Proceedings of the 30th International Conference of the Association for Computer-Aided Architectural Design Research in Asia (CAADRIA) 2025, 141–150.

Touvron, H., T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. 2023. "LLaMA: Open and Efficient Foundation Language Models." arXiv.

Tran, S. V.-T., J. Yang, R. Hussain, N. Khan, E. C. Kimito, A. Pedro, M. Sotani, U.-K. Lee, and C. Park. 2024. "Leveraging large language models for enhanced construction safety regulation extraction." Journal of Information Technology in Construction, 29: 1026–1038. International Council for Research and Innovation in Building and Construction. https://doi.org/10.36680/j.itcon.2024.045.

Vesse, R., R. Zettlemoyer, K. Ahmed, G. Moore, T. Pluskiewicz, and S. Lang. 2025. "dotNetRDF is a powerful and flexible API for working with RDF and SPARQL in .NET environments." Accessed November 2, 2025. https://github.com/dotnetrdf/dotnetrdf.

W3C. 2014a. "RDF 1.1 Concepts and Abstract Syntax." Accessed January 17, 2026. https://www.w3.org/TR/rdf11-concepts/.

W3C. 2014b. "RDF 1.1 Turtle." Accessed January 17, 2026. https://www.w3.org/TR/turtle/.

Wan, H., J. Zhang, Y. Chen, W. Xu, and F. Feng. 2025. "Exploring Gen-AI applications in building research and industry: A review." Building Simulation, 18 (6): 1251–1273. https://doi.org/10.1007/s12273-025-1279-x.

Wang, L., C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei, and J. Wen. 2024. "A survey on large language model based autonomous agents." Frontiers of Computer Science, 18 (6): 186345. https://doi.org/10.1007/s11704-024-40231-1.

Wang, X., and N. El-Gohary. 2023. "Deep learning-based relation extraction and knowledge graph-based representation of construction safety requirements." Automation in Construction, 147: 104696. Elsevier BV. https://doi.org/10.1016/j.autcon.2022.104696.

Wang, Z., R. Sacks, and T. Yeung. 2022. "Exploring graph neural networks for semantic enrichment: Room type classification." Automation in Construction, 134: 104039. Elsevier BV. https://doi.org/10.1016/j.autcon.2021.104039.

Yang, M., Q. Zhao, L. Zhu, H. Meng, K. Chen, Z. Li, and X. Hei. 2023. "Semi-automatic representation of design code based on knowledge graph for automated compliance checking." Computers in Industry, 150: 103945. Elsevier BV. https://doi.org/10.1016/j.compind.2023.103945.

Ying, H., and R. Sacks. 2024. "From Automatic to Autonomous: A Large Language Model- driven Approach for Generic Building Compliance Checking."

Zhang, J., and N. El-Gohary. 2016a. "An Automated Relationship Classification to Support Semi-Automated IFC Extension." In: Construction Research Congress 2016, 829–838. San Juan, Puerto Rico: American Society of Civil Engineers.

Zhang, J., and N. M. El-Gohary. 2015. "Automated Extraction of Information from Building Information Models into a Semantic Logic-Based Representation." In: Computing in Civil Engineering 2015, 173–180. Austin, Texas: American Society of Civil Engineers.

Zhang, J., and N. M. El-Gohary. 2016b. "Extending Building Information Models Semiautomatically Using Semantic Natural Language Processing Techniques." Journal of Computing in Civil Engineering, 30 (5). American Society of Civil Engineers (ASCE). https://doi.org/10.1061/(asce)cp.1943-5487.0000536.

Zhang, Z., C. Zhao, D. Li, P. Li, Y. Chen, H. Zhu, and D. Han. 2026. "Human-in-the-Loop Semantic Rule Base Generation and Dynamic Updating for Automated BIM Compliance Checking: A Knowledge Graph Approach." Buildings, 16 (4): 719. https://doi.org/10.3390/buildings16040719.

Zhao, W. X., K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, and J.-R. Wen. 2023. "A Survey of Large Language Models." arXiv.

Zheng, C., S. Wong, X. Su, Y. Tang, A. Nawaz, and M. Kassem. 2025a. "Automating construction contract review using knowledge graph-enhanced large language models." Automation in Construction, 175: 106179. Elsevier BV. https://doi.org/10.1016/j.autcon.2025.106179.

Zheng, J., and M. Fischer. 2023. "Dynamic prompt-based virtual assistant framework for BIM information search." Automation in Construction, 155: 105067. Elsevier BV. https://doi.org/10.1016/j.autcon.2023.105067.

Zheng, Z., J. Han, K.-Y. Chen, X.-Y. Cao, X.-Z. Lu, and J.-R. Lin. 2025b. "Translating regulatory clauses into executable codes for building design checking via large language model driven function matching and composing." Engineering Applications of Artificial Intelligence, 163: 112823. https://doi.org/10.1016/j.engappai.2025.112823.

Zheng, Z., Y.-C. Zhou, X.-Z. Lu, and J.-R. Lin. 2022. "Knowledge-informed semantic alignment and rule interpretation for automated compliance checking." Automation in Construction, 142: 104524. Elsevier BV. https://doi.org/10.1016/j.autcon.2022.104524.

Zhong, B., X. Xing, H. Luo, Q. Zhou, H. Li, T. Rose, and W. Fang. 2020. "Deep learning-based extraction of construction procedural constraints from construction regulations." Advanced Engineering Informatics, 43: 101003. Elsevier BV. https://doi.org/10.1016/j.aei.2019.101003.

Zhou, P., and N. El-Gohary. 2014. "Semantic-based Text Classification of Environmental Regulatory Documents for Supporting Automated Environmental Compliance Checking in Construction." In: Construction Research Congress 2014, 897–906. Atlanta, Georgia: American Society of Civil Engineers.

Zhu, J., H.-Y. Chong, H. Zhao, J. Wu, Y. Tan, and H. Xu. 2022. "The Application of Graph in BIM/GIS Integration." Buildings, 12 (12): 2162. https://doi.org/10.3390/buildings12122162.

Zhu, S., J. Zhou, L. Cheng, X. Fu, Y. Wang, and K. Dai. 2025. "Research on a BIM Model Quality Compliance Checking Method Based on a Knowledge Graph." Journal of Computing in Civil Engineering, 39 (1). American Society of Civil Engineers (ASCE). https://doi.org/10.1061/jccee5.cpeng-5950.

Zhu, Y., G. Zhao, H. Liu, D. Sun, and Y. He. 2024. "Refining Bridge Engineering-based Construction Scheme Compliance Review with Advanced Large Language Model Integration." In: Proceedings of the 2024 8th International Conference on Big Data and Internet of Things, 297–305. Macau China: ACM.

APPENDIX A: PATH A - REQUIREMENT MAPPING AND CLASSIFICATION PROMPT

This appendix presents the mapping prompt used to classify user requests as either matching existing Knowledge Graph rules or requiring novel code generation. The prompt incorporates chain-of-thought reasoning and prioritizes rule reuse over novel generation.

Component	Specification and Logical Instructions
System Role	Expert Architectural Assistant whose goal is to map the user requirement to an existing Knowledge Graph rule or mark it as novel.
Input Context	(1) The natural-language requirement. (2) A relevant Revit category hint when available (e.g. 'Walls', 'Rooms'). (3) A list of summaries of existing KG rules retrieved as context.
Decision Hierarchy (Step 1: Reuse)	The LLM is required to prioritize reuse. It scans KG summaries for a rule whose Pattern field starts with METHOD: whose Pattern is CustomCSharpFromKG . If a functional match is found, the LLM must reuse the existing KG rule, set IsNovel to false, and set MatchedKgRuleId to the matched rule's ID, even if the user's wording or threshold values differ.
Decision Hierarchy (Step 2: Novelty)	A requirement is declared novel only if no functional match exists in the KG. In that case IsNovel is set to true and MatchedPattern to null.
Element Applicability	The LLM must identify Revit element scope and populate ApplicableCategoryStrings (an array of BuiltInCategory enum names) and/or ApplicableStructuralTypeStrings (an array of StructuralType enum names).
Regulation Traceability	Classification Logic: If a novel requirement does not explicitly mention an official code (e.g., IBC, ACI), it must be classified as a "Custom/User-Defined Regulation" to maintain the integrity of the KG.
Space-Function Filtering	When a room or space type is mentioned (e.g., 'bathrooms', 'bedrooms', 'offices'), the LLM extracts it as a SpaceFunction parameter under ExtractedInputs. If the user says 'all rooms' or specifies no room type, SpaceFunction is set to 'all' or omitted.
Regulation Classification	For novel requirements where the user did not explicitly cite an official code, the LLM must classify the regulation as 'Custom/User-Defined Regulation'. Official code references (e.g., IBC, ACI) may be used only when the existing KG rule already references one or when the user explicitly cites the code.
Reasoning Field	The LLM must populate a mandatory reasoning field justifying why a rule was reused or declared novel, supporting human-in-the-loop review.
Output Format	A single valid JSON object containing MatchedPattern, MatchedKgRuleId, IsNovel, ExtractedInputs (parameters including thresholds, Operator, Unit, and SpaceFunction), ApplicableCategoryStrings and/or ApplicableStructuralTypeStrings, the three conceptual-link arrays (SuggestedConceptUrisOrLabels, SuggestedRegulationUrisOrLabels, SuggestedPropertyUrisOrLabels), the reasoning field, and the validation-metadata fields InterpretedRequirement, NeedsUserConfirmation.

APPENDIX B: PATH B - C# CODE GENERATION PROMPT

This appendix contains the C# code generation prompt that guides the LLM in producing executable compliance-checking logic. The prompt classifies requirements by complexity, injects worked examples retrieved from the Knowledge Graph, and specifies coding constraints for implementing the IComplianceCheckRule interface within the Revit API environment.

Component	Specification and Logical Instructions
System Role	Expert C# developer specializing in the Autodesk Revit API. The task is to generate a complete C# class implementing the IComplianceCheckRule interface in the framework's Generated namespace.
Input Context	(1) Rule metadata: rule ID, natural-language description, and extracted check-logic description, plus the list of Revit parameters or methods the rule needs. (2) Technical references: the interface and ComplianceResult class definitions, the available helper libraries, and the core domain concepts retrieved from the KG.
Complexity Classification	The LLM is instructed to classify the requirement before generating code: Simple: single-element, single-property checks (about 5 to 15 lines of direct logic). Examples: room area threshold, column depth, fire-rating presence. Spatial: checks involving spatial relationships, geometry, or interactions between multiple elements (about 30 to 80 lines of multi-strategy reasoning). Examples: clearance, alignment.
Few-Shot KG	Up to two semantically related KG rules are retrieved (via GetRelevantKgExamplesForPrompt) and injected as worked examples, each including the rule description, target Revit parameters, and the validated CustomCSharpImplementation body. When no semantically related examples are available, the framework falls back to a hand-authored skeleton class hard-coded into the prompt-builder, which illustrates the correct interface implementation, type-parameter access pattern, and ComplianceResult formatting.
Helper Library Usage	The prompt instructs the LLM to use RevitParameterHelper (e.g., GetSmartNumericValue, GetSmartMaterialName) and ArchitecturalElementHelper (e.g., GetSpaceArea, GetSmartSpaceFunction, SquareFeetToSquareMeters) for parameter access, space-function filtering, and unit conversion.
Compliance Result Severity	Each ComplianceResult must be assigned one of four severity levels that govern downstream pass/fail aggregation: Info for non-applicable or filtered elements (excluded from metrics), Pass for successful checks, Warning for passed-with-caveats, and Error for failed checks (the default).
Geometric and Operational Reasoning	For movable elements such as doors, the prompt provides explicit guidance on calculating operational zones using trigonometry and arc-sampling rather than simple bounding-box expansion, and on testing conflicts between sampled arc points and the bounding boxes of nearby elements.
Coding Constraints	The prompt prohibits expression-bodied property syntax (=>), requires full property syntax. It instructs the LLM to use string.Format or concatenation when string interpolation produces CS1056 errors, to use .Value rather than the obsolete. IntegerValue on ElementId, and to access type-level parameters (such as section dimensions h, b, Depth, Width) by retrieving the ElementType via doc.GetElement(element.GetTypeId()) rather than from the instance. Generic types must be written with literal < and > rather than Unicode escapes.
Parameter-Aware Logic	Threshold values must be retrieved from the ruleParameters dictionary (with a sensible default if the key is absent) rather than hard-coded, so that the same rule can be reused across thresholds.
Output Format	A single complete C# class file with the necessary using directives and namespace declaration, with no Markdown fencing or commentary outside C# comments.

APPENDIX C: PATH B - C# CODE DEBUGGING PROMPT

This appendix presents the debugging prompt used in the self-correction loop when generated code fails to compile. The prompt feeds compiler error messages and the original code back to the LLM, instructing it to repair specific syntactic and API-level errors while preserving the rule's functional intent.

Component	Specification and Logical Instructions
System Role	Expert C# developer (Revit API) tasked with debugging and correcting code that failed to compile, while preserving the rule's functional intent.
Input Context	(1) The rule metadata (ID, description, logic description, expected Revit parameters, and applicable concepts/regulations from the KG). (2) The interface and result-class definitions for reference. (3) The erroneous C# code. (4) The compiler error messages with line and column numbers.
Targeted Error-Correction Guidance	The prompt enumerates the syntactic and API errors the LLM must address. The most heavily emphasized is CS1056 ('Unexpected character $'), where the prompt instructs the LLM to rewrite affected lines using string.Format or simple concatenation if $"{...}" interpolation has failed in earlier attempts. Other errors covered include: replacing the obsolete ElementId.IntegerValue with .Value (CS0618); correcting type-parameter access (CS1061) by retrieving the ElementType via doc.GetElement(element.GetTypeId()) and looking up parameters such as h, b, Depth, or Width on the type rather than the instance; fixing mismatched braces, parentheses, and brackets; restoring missing semicolons (CS1002); replacing expression-bodied property syntax with full property syntax; correcting invalid identifiers and using directives (CS0103, CS0234); and ensuring generic types use literal < and > rather than Unicode escapes.
Category-Specific Parameter Guidance	When applicable Revit categories are known, the prompt injects category-specific parameter naming guidance (for example, common parameter patterns for windows, rooms, and structural elements), and instructs the LLM to use exact Revit UI parameter names rather than invented descriptive names.
Coding Constraints (Reaffirmed)	The same constraints from the generation prompt are repeated: no expression-bodied properties, robust parameter access with null and HasValue checks, correct unit conversions, complete ComplianceResult population (including RuleId, RuleDescription, ActualValue, ExpectedValue, and Unit where applicable), and graceful handling of inapplicable elements.
Output Format	The fully corrected C# class, returned as code only, with no Markdown fencing or commentary outside C# comments.