2026 European Conference on Computing in Construction
Corfu, Greece
July 12–15, 2026

ASSESSING THE VIABILITY OF LLM AGENTS FOR GENERATING REUSABLE COMPLIANCE CHECKING FUNCTIONS

Stefan Fuchs[1,2], Sylvain Hellin[1,2], and André Borrmann[1,2]
[1]Chair of Computing in Civil and Building Engineering, Technical University of Munich, Munich, Germany
[2]TUM Georg Nemetschek Institute, Technical University of Munich, Munich, Germany

Abstract

The complexity of building codes and variations in Building Information Modelling (BIM) practices have long challenged Automated Compliance Checking (ACC). The success of Large Language Models (LLMs) in code generation opens new opportunities for this domain. Unlike previous work relying on machine-readable rules or LLMs using hard-coded tools that limit scalability, this paper examines whether LLMs, using the Code-Act pattern, can generate scalable compliance logic for regulations up to Solihin and Eastman's complexity class 3. We show that iterative refinement with a strong verifier evolves reusable checking functions for consistently modelled BIM data, while geometric reasoning requiring auxiliary constructions remains challenging.[1]

Introduction

The digitalization of building permit processes promises faster, more consistent reviews of construction projects. Accelerating these processes is particularly relevant given the pressing demand for housing in many regions, where lengthy approval cycles contribute to construction delays, and increased costs and uncertainty (Bloch and Fauth, 2023). At the center of this effort lies ACC, which refers to the automatic verification of building designs against applicable codes and standards. If realized effectively, ACC offers a dual benefit: architects can pre-check their building models prior to submission, leading to higher-quality designs, while authorities can review submissions faster and more thoroughly.

Eastman et al. (2009) proposed a widely adopted four-step process for automated rule-based checking: (1) rule interpretation, which involves translating regulatory text into a machine-readable format; (2) building model preparation, including semantic enrichment to make information that is implicitly present in the model explicitly available; (3) rule execution, encompassing spatial checks, simple property checks, and more complex calculations; and (4) reporting of compliance results.

The ACC process has seen substantial evolution over time. For example, for rule interpretation, early approaches relied on hard-coded implementations, resulting in opaque black-box systems that were difficult to modify or extend (Solihin, 2004). Subsequent work shifted toward machine-readable intermediate representations, such as the RASE (Requirement, Applicability, Selection, and Exception) semantic markup methodology (Hjelseth and Nisbet, 2011). Such representations offer greater transparency but prove difficult to author and often fail to capture the full complexity of regulatory language. Natural language processing (NLP) techniques were applied to assist this translation, initially, through rule-based methods that required considerable manual effort and were difficult to scale (Zhang and El-Gohary, 2017). Machine learning and deep learning approaches improved scalability across different types of regulatory texts but introduced a dependency on high-quality, labeled training data (Fuchs et al., 2024). LLMs have further reduced data requirements and enabled human-in-the-loop workflows, for example, to apply RASE markup utilising YAML-based representations (Al-Turki et al., 2024). More recent work has explored feeding regulations directly into LLMs for ACC, bypassing intermediate representations entirely (Iversen and Huang, 2026). However, most LLM agent-based approaches required manually authored tool functions that the LLM can invoke, a process that reintroduces significant manual effort. Allowing LLMs to generate the checking code themselves addresses this limitation, but the resulting functions have generally not been reusable across different building models (Madireddy et al., 2025). Achieving reusability would make these functions independently deployable as transparent, reproducible compliance checks, shareable as plain code across practitioners and tools.

In this paper, we assess whether LLM agents can generate reusable compliance checking functions for building regulations with complexity up to Class 3 in the classification by Solihin and Eastman (2015), which require extended data structures such as geometric helper elements. The LLM agent uses the Code-Act architecture (Wang et al., 2024), where the agent interleave reasoning with code generation and execution. By employing a strong verifier, i.e., ground-truth compliance results for multiple Industry Foundation Classes (IFC) models against which generated functions are iteratively tested, we investigate whether this process enables the development of verifiable, deterministic, and reusable compliance checking functions. Accord-

[1]Source code and data available at: `https://github.com/stefan-1992/ACC-function-generation`

ingly, we address the following research questions (RQs):

- **RQ1:** To what extent can LLM agents generate reusable compliance checking functions that generalize across building models?
- **RQ2:** What difficulties arise when scaling across diverse building models?
- **RQ3:** Up to what regulation complexity class can LLM agents reliably generate checking functions?

These questions are investigated through a systematic training and evaluation framework applied to 16 compliance rules across 12 building models. The generated functions are evaluated for correctness and generalization, with a detailed analysis of failure modes and the impact of regulation complexity.

Related work

The challenge of translating building regulations into executable checks has been addressed through two contrasting paradigms. Deterministic, machine-readable representations offer transparency and reproducibility but require substantial manual authoring effort and struggle to capture the full complexity of regulatory language. For example, Solibri, one of the most used commercial tools, provides significant functionality for general model checking and compliance checking. Existing rule templates can be parametrised, but checks beyond the available templates require custom Java development through Solibri's API. The inner workings of built-in rules remain a black box, and the proprietary format makes keeping rules up-to-date with regulatory amendments difficult (Solibri Inc., 2026). Moreover, many regulations cannot easily be transformed into parameterised rule templates or other machine-executable formats, particularly when they involve complex spatial relationships or context-dependent conditions (Hjelseth and Nisbet, 2011). Information availability poses an additional challenge: approaches such as Singapore's IFC-SG standard (Government of Singapore, 2026) specify precise information requirements, validated through Information Delivery Specifications (IDS), but this shifts the burden onto the modeller. Semantic enrichment can partially alleviate this burden by automatically deriving missing information from the model (Bloch, 2022), yet it remains an open research problem. LLM-based approaches offer a more dynamic alternative that can work directly with natural language regulations and heterogeneous model data, yet they often lack the determinism and reusability needed for production use.

Agent-based approaches to automated compliance checking

Several recent studies have explored LLM-based agents for ACC. Ying and Sacks (2024) proposed a framework with different LLM techniques, such as Retrieval-Augmented Generation (RAG), ReAct (reasoning–action) agents, and tool usage, which cover the full ACC pipeline: retrieving relevant regulations from PDF documents, executing compliance checks through switching between reasoning and action phases with predefined tool functions, and resolving non-compliance issues. While this approach demonstrates the feasibility of end-to-end agent-based checking, it relied on manually authored tool functions, and the compliance checking was only tested on one building requirement.

Zheng et al. (2026) and Iversen and Huang (2026) showed that LLMs can identify which predefined functions to orchestrate for a given rule check, but in Iversen and Huang (2026) only a limited number of functions were actually implemented, restricting the approach's practical applicability. While Zheng et al. (2026) used LLMs for function implementation in a case study, they did not systematically evaluate the LLMs' capability for this task. Shi et al. (2025) explored fine-tuning LLMs specifically for automated code compliance, indicating that domain adaptation can improve checking accuracy. However, they did not generate fully executable functions but high-level function calls, which removes the LLM's ability to receive feedback from the rule execution and adjust the generated functions accordingly. Madireddy et al. (2025) advanced the concept by having LLMs generate Python scripts for compliance checking; however, the scripts required manual transfer to Revit and human-in-the-loop verification, limiting automation. Furthermore, several rule-specific hints were required, implying extensive manual analysis.

BIM question answering as a foundation

A related field is BIM question answering, where LLM agents generate code to extract information from building models. Zheng and Fischer (2023) introduced BIMS-GPT, a virtual assistant that dynamically generates queries against BIM data. However, this approach is limited to information directly available in the BIM database. Hellin et al. (2025) developed a multi-agent system for natural language-based BIM question answering. The agents can invoke multiple IfcOpenShell-based tools to explore the BIM model until it has enough information gathered to answer the question. Nithyanantham et al. (2025) proposed MCP4IFC (Model Context Protocol), providing tools to query, edit, and create IFC elements, including dynamic code generation when the existing toolkit is insufficient. Li et al. (2025) demonstrated an LLM agent capable of retrieving data from multiple sources and generating code for spatial queries on building models. These systems address similar technical challenges to ACC, i.e., interpreting domain-specific queries and invoking functions or generating executable code, but typically evaluate results on individual models, where the logic can be adapted to the model at hand. Furthermore, the geometric complexity of such queries is typically lower than in ACC.

Our work extends tool generation to compliance checking functions that must generalize across multiple building models, necessitating a more scalable evaluation framework. Crucially, LLMs are used only during function generation, not at inference time: the resulting checking functions are deterministic, reusable, and execute without LLM

involvement, combining the authoring flexibility of LLMs with the auditability of conventional rule logic.

Methodology

We adopt a machine learning-inspired training strategy in which LLM agents iteratively write, refine, and validate compliance checking functions operating on BIM models in IFC format (Figure 1), which are then tested on unseen models. This strategy is inspired by the evolutionary code optimization strategy of AlphaEvolve (Novikov et al., 2025), where a strong verifier guides iterative improvement. At the center of the approach lies the function creation, which draws on the Code-Act pattern to iteratively generate and refine checking functions.

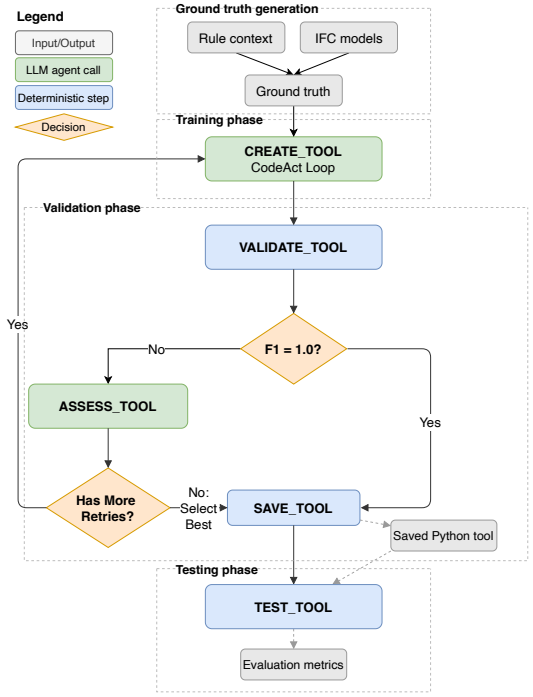

Figure 1: Overview of the agent pipeline for generating compliance checking functions.

Ground truth generation

While in AlphaEvolve, a test function was used to evaluate the generated code, our framework relies on ground-truth compliance results for each building model. These can be produced manually, through coded reference implementations, or using existing rule-checking tools, as long as each element receives a definitive pass or fail label.

The dataset should comprise at least three building models to support separate training, validation, and test splits, with violations present across most or all models, and a mix of compliant and non-compliant elements. Having more BIM models with rule violations increases generalisability of the generated functions, as the agent must handle diverse geometric and semantic configurations rather than overfitting to a single case. We use the Globally Unique Identifier (GUID) to identify and compare violating building elements, producing a deterministic binary signal per element that remains stable throughout the experiment.

Training phase

The training phase begins by selecting a *primary* training BIM model, selected as the model containing the highest number of non-compliant issues, to provide the most informative training model as context for testing the developed functions. The code-generation agent (CREATE_TOOL) then receives the natural-language rule text, including any relevant thresholds and exceptions, along with the expected answers for each training BIM model (Figure 2). It synthesizes a candidate checking function in Python, using `IfcOpenShell` for IFC access and `trimesh`/`shapely` for geometric operations where needed.

Rule: This rule checks that doors and windows in the model are located in the same floor as the wall they are related to. The rule also checks that the model does not contain any orphan doors or windows (a door or window without a relation to any wall).

Parameters: Check Orphan Doors and Windows: True

Question: What doors or windows in the current IFC model are not located in the same floor as the wall they are related to, or are orphan doors/windows without a relation to any wall? Return a list of IFC GUIDs of all elements that violate this rule.

Ground Truth:

`Model: 146` – pass (no violations, return `[]`)

`Model: duplex (primary)` – 2 violations:

Required GUIDs: `[1hOS...lSFK, 1hOS...lSDm]`

Issue 1: *''Door.1.7 And Wall.0.10'': Door components are included in Level 2, but Wall components are included in Level 1. Door components should be included in the same floor as the component they belong to.*

Issue 2: *''Door.1.8 And Wall.0.13'': [...]*

Figure 2: Condensed example of the agent prompt for the ''doors and windows'' rule.

During the code-generation phase, visualised in Figure 3, the agent operates in a CodeAct loop of up to $n_{\text{max\_iter}}$ iterations. The agent's task is to iteratively develop a function that identifies the same violations as the ground truth for all training models. At each iteration, it returns either a *CodeAction* or the final *ACC Function*. The *CodeAction* contains reasoning traces and Python code that is executed in a Python sandbox where imports and runtime tools are pre-loaded. Multiple BIM models are used for training to encourage generalization from the outset, ensuring that the function does not overfit to a single building's characteristics. The agent commonly developed strategies to first test functions on only the primary model, before including test cases for all training models. The agent can decide at any point to return the final implementation of the *ACC Function*, which terminates the loop. On the last permitted iteration (i.e., $n_{\text{max\_iter}}$), the agent is prompted to return its best implementation. Before proceeding to validation, a pre-check executes the function against all training models to verify the absence of runtime errors; if it crashes, the agent re-enters the loop with the error log appended (up to three consecutive pre-check failures were permitted).

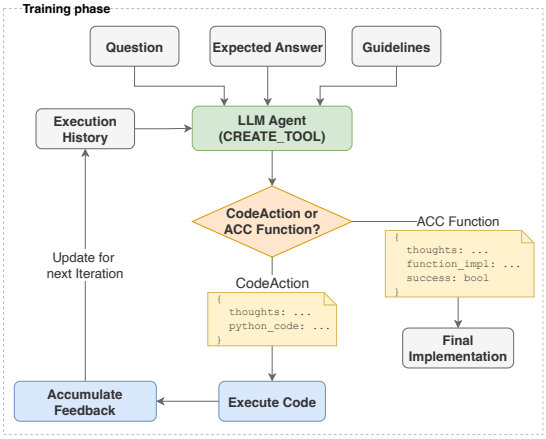

Figure 3: Overview of the code act training loop for generating compliance checking functions.

The agent's prompt specifies a function contract; the generated function must accept a single IFC file path and return a list of non-compliant element GUIDs. Beyond the geometric libraries, the agent has access to `query_ifcopenshell_docs` as runtime tool, which retrieves up-to-date IfcOpenShell documentation and code examples for a given topic.

Validation phase

Once the agent has converged on a function during training, the function is executed against a separate set of validation building models. A dedicated assessment agent interprets the training and validation results. This agent receives the tool's source code, its execution logs, aggregated performance metrics (true-positive, false-positive, and false-negative counts), and a comparison of expected versus predicted element counts. We use an LLM because the diagnosis requires correlating source code logic with per-model error distributions, which involves too many interacting factors for simple heuristic rules. The agent determines the likely cause of failure from five diagnostic categories (overfitting, missing generalisation, implementation bug, model difference, or unknown) and outputs a concrete, GUID-free improvement hint for the code-generation agent. Critically, the assessment agent is prompted not to disclose ground-truth results from the validation set to the code-generation agent; only this high-level diagnostic feedback is provided, discouraging information leakage. Exposing validation GUIDs would allow the code-generation agent to reverse-engineer element-specific properties, or worse, hard-code the GUIDs, undermining generalisation beyond the training distribution.

The assessment result is checked against a retry budget (default: 2 retries). If retries remain and the diagnosis suggests a fixable error, the improvement hint is injected into the next CREATE_TOOL prompt. If the budget is exhausted or the error is deemed negligible, the best-performing implementation is saved. This mechanism bounds computational cost while still allowing targeted refinement.

Testing phase

Functions that pass validation are evaluated on a held-out test set of building models, measuring how well the generated functions transfer to unseen buildings. The separate test set ensures that neither the agent's prompts nor the tooling are optimised against it. We apply this process to 16 compliance rules spanning multiple complexity classes as detailed in the next section.

Experimental setup

We evaluate our approach using 12 building models, divided equally into training, validation, and test models (4 models each), as shown in Table 1. All models are freely available: three originate from the GNI BIM Dataset - 2025_BIMfundamentals (Wang et al., 2026), and the remaining models are sourced from IFCBench (Hellin et al., 2025). The data split ensures that each compliance rule exhibits non-compliant elements in the training set, with test and then validation sets prioritised where possible.

Table 1: Building models used in each data split.

Training	Validation	Test
146*	106*	4351
AC20	172*	Digital Hub
Dental Clinic	FZK House	S. MacAlister
Duplex	Smiley West	WBDG Office

*From GNI BIM Dataset (Wang et al., 2026).

Ground-truth compliance labels are produced using Solibri (Solibri Inc., 2026). We executed predefined rule sets against each IFC model using Solibri's autorun. From the resulting BIM Collaboration Format (BCF) reports, we identified issues with single (or well separable) constraints and defined how to automatically locate them and distinguish the GUIDs that violate the rule from contextual reference elements. Where rules depend on semantic classifications (e.g., space usage types), we applied them through importable Solibri pattern rules. To keep the agent on equal footing with the verifier, we provided it with the `classify_spaces` tool, which assigns semantic labels (e.g., circulation, habitable) to IFC spaces by matching names against the predefined patterns.

Compliance rules

We select 16 compliance rules spanning the complexity classes defined by Solihin and Eastman (2015), summarized in Table 2. These range from simple property checks (Class 1) through rules requiring derived attributes or geometric relationships among multiple objects (Class 2) to rules requiring extended data structures and geometric helper elements (Class 3). In practice, Class 1 and 2 rules may behave as higher-complexity checks when the required properties or relations are absent from the IFC model and must be derived geometrically.

Table 2: Compliance rules grouped by Solihin & Eastman complexity class, with non-compliant element counts across data splits.

Rule	Class	Ruleset	Description	Train	Val	Test
Slab thickness	1	BIM Validation	Slab thickness within 0.30 m–1.0 m range	14	–	3
Riser height	1	ADA/ABA - 504.2	Risers between 100 mm and 180 mm high	5	37	3
Tread length	1	ADA/ABA - 504.2	Treads at least 280 mm deep	2	45	2
Non-uniform risers/treads	2	ADA/ABA - 504.2	Uniform riser heights and tread depths per flight	2	21	–
Stair–slab connection	2	ADA/ABA - 504.2	All stairs connected to slabs	3	–	2
Doors and windows	2	BIM Validation	Doors/windows on same floor as host wall; no orphans	7	8	18
Large spaces ≥ 2 doors	2	Adv. Space Check	Spaces ≥ 50 m^2 have at least 2 doors	22	9	25
Same-storey elevation	2	Gen. Space Check	Spaces in same storey share bottom elevation	180	110	124
Space validation (inside)	2	Gen. Space Check	No components incorrectly inside spaces	17	7	17
Space validation (intersect)	2	Gen. Space Check	Spaces do not intersect with slabs/walls	24	55	22
Circular space	3	ADA/ABA - 304.3.1	Wheelchair turning space ≥ 1525 mm diameter	2	22	2
Clear floor space	3	ADA/ABA - 305.3	Clear floor/ground ≥ 760 × 1220 mm	18	23	1
Two doors in series	3	ADA/ABA - 404.2.5	≥ 1220 mm between hinged/pivoted doors in series	8	–	2
Clearance front of doors	3	BIM Validation	Sufficient clearance on both sides of doors	28	30	11
Slabs guarded against falling	3	BIM Validation	Slab edges protected by barriers or acceptable drop	34	11	8
Unallocated areas	3	BIM Validation	Floor area must be assigned to a space ≤ 0.50 m^2	6	–	23

Evaluation metrics

Evaluation is performed at the building element level using unique element identifiers (i.e., GUIDs). For each model, we compare the set of GUIDs returned by the generated ACC function against the ground-truth set. For violations involving multiple elements, each GUID is counted once. We compute precision, recall, and F1 score per model. The *average* F1 score ($F1_{\text{avg}}$) is the arithmetic mean of the per-model F1 scores (macro-average over models). As a special case, if a model is fully compliant (no non-conforming elements) and the function correctly identifies it as such, this yields Precision, Recall, and F1 score of 1.0. As an alternative, the *aggregated* F1 score ($F1_{\text{agg}}$) is computed by pooling all true-positive, false-positive, and false-negative counts across models before calculating precision, recall, and F1 score (micro-average over models). This provides a more balanced measure, as fully compliant models receive proportionally less weight compared to $F1_{\text{avg}}$. In Table 3, per-class and overall averages are macro-averaged over the rules within each class and over all 16 rules, respectively.

Agent configuration

The agents are powered by GLM-4.7 by Z.ai[2], chosen for its competitive performance on SWE-Bench Verified[3] and favourable performance-to-cost ratio. The function creation agent is permitted up to 15 internal coding iterations per attempt ($n_{\text{max\_iter}} = 15$). The validation loop allows up to 2 feedback-driven retries before the best implementation is saved. The agent has access to `IfcOpenShell` for IFC model traversal, and `shapely` and `trimesh` for 2-D and 3-D geometric operations, respectively. The libraries are preloaded and briefly described within the prompt. Documentation retrieval is currently only supported for IfcOpenShell; extending this to cover `trimesh` and `shapely` is left for future work.

[2]https://z.ai/blog/glm-4.7

[3]https://openai.com/index/introducing-swe-bench-verified/

Results

Table 3 summarises the per-rule performance across the three data splits. We use the aggregated F1 score ($F1_{\text{agg}}$) as the primary evaluation metric. Overall, the LLM agent shows moderate capability for implementing ACC functions, with $F1_{\text{agg}}$ values of 0.55 on training, 0.54 on validation, and 0.40 on test. The near-identical training and validation scores suggest that the validation feedback loop generalises to the validation models. The subsequent drop on the unseen test set, however, indicates that many generated functions still overfit to assumptions about how geometry, properties, and relationships are represented in the training and validation IFC models. Performance varies by complexity class. Class 1 rules achieved near-perfect validation scores ($F1_{\text{agg}} = 0.96$), yet dropped sharply on the test set (0.38). Class 2 rules achieved the strongest test results ($F1_{\text{agg}} = 0.57$). The strong Class 2 average is driven by three rules that generalised fully to the test set, whereas the weak Class 3 average (0.21) reflects the continued difficulty of rules that require auxiliary geometric constructions or richer rule logic.

Rules that achieved perfect or near-perfect training scores showed varying degrees of degradation on test, from none (*doors and windows*, *same-storey elevation*, and *stair–slab connection*: $1.00 \rightarrow 1.00$) to complete failure (*slab thickness*: $1.00 \rightarrow 0.00$). The validation feedback loop improved some rules: for *tread length*, the validation $F1_{\text{agg}}$ reached 0.89, and the function still achieved 0.67 on test. For more complex rules, however, the feedback loop could not compensate for missing geometric detail, incomplete IFC relationships, or ambiguous rule semantics. The gap between $F1_{\text{avg}}$ and $F1_{\text{agg}}$ on the test set (0.59 vs. 0.40) suggests that performance remains uneven across models, with several rules receiving inflated average scores because some test models are fully compliant.

Table 4 shows the execution metrics for each rule. Total duration ranges from 4.3 minutes (slab thickness) to 56.9 minutes (clear floor space). Token usage spans 116 k

Table 3: Per-rule $F1$ scores across data splits. $F1_{\text{avg}}$: macro-average over models; $F1_{\text{agg}}$, P_{agg}, R_{agg}: micro-average over models (pooled counts). Class and overall rows: macro-average over rules.

		Training				Validation				Test			
Rule	**Cl.**	$F1_{\text{avg}}$	P_{agg}	R_{agg}	$F1_{\text{agg}}$	$F1_{\text{avg}}$	P_{agg}	R_{agg}	$F1_{\text{agg}}$	$F1_{\text{avg}}$	P_{agg}	R_{agg}	$F1_{\text{agg}}$
Slab thickness	1	1.00	1.00	1.00	1.00	1.00†	1.00†	1.00†	1.00†	0.75	0.00	0.00	0.00
Riser height	1	0.75	1.00	0.67	0.80	1.00	1.00	1.00	1.00	0.54	0.30	1.00	0.46
Tread length	1	0.75	0.33	1.00	0.50	0.50	1.00	0.80	0.89	0.92	0.50	1.00	0.67
Class 1 avg.	1	*0.83*	*0.78*	*0.89*	*0.77*	*0.83*	*1.00*	*0.93*	*0.96*	*0.74*	*0.27*	*0.67*	*0.38*
Non-uniform risers/treads	2	1.00	1.00	1.00	1.00	0.50	0.00	0.00	0.00	0.75†	0.00†	0.00†	0.00†
Stair–slab connection	2	1.00	1.00	1.00	1.00	1.00†	1.00†	1.00†	1.00†	1.00	1.00	1.00	1.00
Doors and windows	2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Large spaces ≥ 2 doors	2	0.62	0.64	0.32	0.42	0.75	1.00	0.89	0.94	0.14	0.80	0.16	0.27
Same-storey elevation	2	1.00	1.00	1.00	1.00	0.75	1.00	0.24	0.38	1.00	1.00	1.00	1.00
Space validation (inside)	2	0.57	0.43	0.18	0.25	0.50	0.00	0.00	0.00	0.40	0.60	0.18	0.27
Space validation (intersect)	2	0.20	0.21	0.88	0.34	0.25	0.28	0.35	0.31	0.50	0.30	1.00	0.46
Class 2 avg.	2	*0.77*	*0.75*	*0.77*	*0.72*	*0.68*	*0.61*	*0.50*	*0.52*	*0.68*	*0.67*	*0.62*	*0.57*
Circular space	3	0.75	0.40	1.00	0.57	0.75	1.00	0.05	0.09	1.00	1.00	1.00	1.00
Clear floor space	3	0.50	0.00	0.00	0.00	0.25	0.00	0.00	0.00	0.50	0.00	0.00	0.00
Two doors in series	3	0.50	0.00	0.00	0.00	1.00†	1.00†	1.00†	1.00†	0.75	0.00	0.00	0.00
Clearance front of doors	3	0.25	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Slabs guarded against falling	3	0.45	0.51	0.87	0.64	0.23	0.04	0.40	0.08	0.23	0.21	0.43	0.29
Unallocated areas	3	0.35	0.20	0.33	0.25	1.00†	1.00†	1.00†	1.00†	0.00	0.00	0.00	0.00
Class 3 avg.	3	*0.47*	*0.18*	*0.37*	*0.24*	*0.62*	*0.51*	*0.41*	*0.36*	*0.41*	*0.20*	*0.24*	*0.21*
Overall avg.	1-3	**0.67**	**0.55**	**0.64**	**0.55**	**0.69**	**0.65**	**0.54**	**0.54**	**0.59**	**0.42**	**0.49**	**0.40**

† All models were compliant; the scores reflect false-positive control only.

(slab thickness) to 2.4 M (clearance front of doors), with geometry-heavy Class 2 and 3 rules consistently consuming the most time and tokens. *Doors and windows* converged quickly and achieved perfect scores across all three splits, suggesting that the agent performs well when the rule can be expressed through relatively direct relationship checks. By contrast, *clear floor space* (56.9 min), *space validation (intersect)* (45.4 min), and *clearance front of doors* (45.3 min) were among the longest-running rules, reflecting the cost of repeated geometric exploration without strong generalisation.

Discussion

The results indicate that reusable compliance checking functions can be generated when IFC semantics are consistent, but generalisation remains fragile across heterogeneous models. The perfect test performance of *doors and windows*, *same-storey elevation*, and *stair–slab connection* shows the approach is viable when the relevant semantics are explicitly represented and stable across models. The broader decline from training to test suggests that many functions encode assumptions about properties, relationships, or geometry that do not transfer reliably. Manual error analysis of the test set explains this gap. Geometry-based heuristics were the dominant failure mode, affecting seven rules: *clearance front of doors*, *slabs guarded against falling*, *space validation (inside)*, *space validation (intersect)*, *non-uniform risers/treads*, *clear floor space*, and *unallocated areas*. The agent often produced plausible but coarse approximations based on bounding boxes, convex hulls, perimeter sampling, or random point placement. Such heuristics succeed on training models but may fail on unseen IFC geometries, where Solibri uses richer geometric reasoning and internal tolerances.

Relationship traversal is a second major source of error. The clearest examples are *large spaces ≥ 2 doors* and *two doors in series*, where the generated functions rely on `IfcRelSpaceBoundary` to associate spaces, walls, and doors. When these relationships are incomplete or absent, the functions fail to identify the relevant door pairs or counts. Property lookup is similarly brittle: *slab thickness* fails because the relevant property is stored under the German label `Dicke`, while *riser height* is affected by inconsistent property interpretation, for example when `Pset_StairCommon.RiserHeight` reports 0.55 m while the Revit-derived maximum riser height is 0.18 m. These examples show that the core challenge is not only code generation, but also learning robust strategies for handling heterogeneous IFC conventions.

Many of these issues could be reduced by increasing the number and diversity of training and validation models. A larger dataset would expose the agent to more variants of relationship structures, property naming conventions, unit systems, and geometric edge cases, making it less likely to overfit to a single export implementation. Realising that benefit would require longer exploration budgets, since the agent would need more iterations to inspect a broader range of failure cases. At the same time, scaling to a broader model set would require automating Solibri object classification, which is currently provided through manually prepared pattern rules. Without such automation, the approach does not scale cleanly to new models

Table 4: Execution metrics per compliance rule: retry count, coding iterations per attempt, and time/token breakdown.

		Iterations per attempt			Agent (min)		Execution (min)			
Rule	**Retries**	**Att_0**	**Att_1**	**Att_2**	**CREATE**	**ASSESS**	**Tr+Val**	**Test**	**Total (min)**	**Tokens**
Slab thickness	0	9	–	–	3.1	0.0	0.8	0.4	4.3	116,497
Riser height	0	14	–	–	11.4	0.0	0.7	0.4	12.5	328,637
Tread length	2	4	6	7	6.5	1.0	2.2	0.4	10.1	163,662
Non-uniform risers/treads	2	9	15	12	14.8	0.2	1.5	0.4	16.9	373,004
Stair–slab connection	2	13	15	7	26.4	2.3	2.4	0.4	31.5	511,586
Doors and windows	0	15	–	–	9.4	0.0	0.8	0.4	10.5	430,186
Large spaces ≥ 2 doors	2	13	15	15	15.6	0.9	2.3	0.4	19.2	920,618
Same-storey elevation	2	12	15	15	17.6	0.4	2.3	0.4	20.7	1,465,624
Space validation (inside)	2	2	15	7	9.8	1.1	3.9	0.6	15.4	579,809
Space validation (intersect)	2	13	15	15	38.4	0.9	5.4	0.6	45.4	1,435,136
Circular space	2	6	11	13	37.4	3.2	2.4	0.4	43.4	697,468
Clear floor space	2	15	12	15	48.2	5.9	2.4	0.4	56.9	1,131,448
Two doors in series	0	15	–	–	10.4	0.0	1.3	0.6	12.3	249,662
Clearance front of doors	2	15	15	15	33.4	1.0	10.5	0.4	45.3	2,378,279
Slabs guarded against falling	2	15	15	15	30.8	0.4	4.2	2.1	37.5	747,190
Unallocated areas	1	15	15	–	11.6	0.3	2.0	0.5	14.3	1,033,179

because both the ground-truth generation and the agent's auxiliary classification tool depend on handcrafted semantic mappings.

Geometric reasoning poses a barrier that current LLMs cannot overcome reliably. The weak Class 3 scores confirm this. The circular space result demonstrates that geometry per se is not the barrier: because fitting a circle inside a polygon requires no model-specific assumptions, the agent achieved a perfect test score ($F1_{\text{agg}} = 1.00$) despite the check's geometric nature. In contrast, rules that require helper elements such as door swing arcs, clearance zones, or barrier envelopes remain difficult, and the resulting implementations often fall back on heuristics that are too coarse to match Solibri consistently.

Another difficulty is that several Solibri checks combine multiple sub-rules within a single template. We partly addressed this by passing rule parameters to the agent, but their exact operational meaning often remains opaque from the prompt alone. This is visible in *space validation (inside)*, where Solibri flags both, components inside spaces and cases where a space does not properly touch the slab or surface below itself. Access to Solibri rule documentation via RAG could help the agent interpret these composite checks and checking parameters more faithfully and reduce the need to infer hidden sub-rules from sparse examples. Without visual feedback on intermediate geometry, the agent cannot verify whether helper elements such as clearance zones are correctly positioned.

We note several limitations. The 12-model evaluation limits statistical robustness. The GUID-level evaluation counts each element once, regardless of its involvement in multiple violations of the same type, potentially overestimating quality for rules that affect many similar elements. The ground truth was established through Solibri rule checks; differences in rule interpretation between the ground truth and the generated ACC functions cannot be excluded. A future dataset should include checks from multiple sources and different methods for generating (or manually creating) the ground-truth violations. Semantic labels for spaces and doors had to be provided via the `classify_spaces` tool, applying pre-defined name-matching patterns. Simple name-matching is effective in controlled settings but produces false positives when naming differs across building types, and introduces a manual dependency that does not scale to new models. Automating object classification is therefore important for scaling the evaluation pipeline and giving the agent access to consistent semantic categories. Integrating more dynamic components, such as semantic similarity or auxiliary agents for edge cases, might improve generalisation but would also weaken the deterministic character of the generated functions, which is central to their reusability across design tools.

Conclusions

This study investigated whether LLM-based agents using the Code-Act paradigm can generate reusable compliance checking functions from building regulations. The results show that such functions are achievable for a subset of Class 1 and Class 2 rules when IFC representations are consistent and the required relationships or properties are explicitly available. However, checks that depend on geometry-heavy heuristics, incomplete relationship traversal, non-standard property lookup, or composite Solibri rule semantics remain difficult, especially for Class 3 rules.

Several directions for future work emerge from these findings. First, larger and more diverse datasets are needed so that the agent can cope with variation in IFC exports, property names, unit conventions, and relationship completeness. Therefore, Solibri object classification should be automated so that semantic labels can be generated consistently at scale rather than supplied through manual pattern rules. Additionally, a curriculum-based generation strategy, in which agents reuse and compose previously veri-

fied functions, might reduce the search space for complex rules. Finally, the generated functions can be integrated into design tools and further enable extended use cases, including reinforcement learning-based design adaptations.

Acknowledgments

We thank the TUM Leonhard Obermeyer Center for financially supporting this research.

References

Al-Turki, D., Hettiarachchi, H., Gaber, M. M., Abdelsamea, M. M., Basurra, S., Iranmanesh, S., Saadany, H., and Vakaj, E. (2024). Human-in-the-loop learning with LLMs for efficient RASE tagging in building compliance regulations. IEEE Access, 12:1–1.

Bloch, T. (2022). Connecting research on semantic enrichment of BIM - review of approaches, methods and possible applications. J. Inf. Technol. Constr., 27(20):416–440.

Bloch, T. and Fauth, J. (2023). The unbalanced research on digitalization and automation of the building permitting process. Adv. Eng. Inform., 58(102188):102188.

Eastman, C., Lee, J.-M., Jeong, Y.-S., and Lee, J.-K. (2009). Automatic rule-based checking of building designs. Autom. Constr., 18(8):1011–1033.

Fuchs, S., Dimyadi, J., Witbrock, M., and Amor, R. (2024). Intermediate representations to improve the semantic parsing of building regulations. Advanced Engineering Informatics, 62:102735.

Government of Singapore (2026). CORENET X Code of Practice. 3.1 edition.

Hellin, S., Nousias, S., and Borrmann, A. (2025). Natural Language Information Retrieval from BIM Models: An LLM-Based Agentic Workflow Approach. In Proc. of the European Conference on Computing in Construction (EC3).

Hjelseth, E. and Nisbet, N. (2011). Capturing normative constraints by use of the semantic mark-up RASE methodology. Proceedings of CIB W78-W102 conference.

Iversen, O. and Huang, L. (2026). Leveraging large language models for BIM-based automated compliance checking. Automation in Construction, 182:106707.

Li, A., Wong, P. K.-Y., Tao, X., Ma, J., and Cheng, J. C. P. (2025). An interactive system for 3D spatial relationship query by integrating tree-based element indexing and LLM-based agent. Adv. Eng. Inform., 66(103375):103375.

Madireddy, S., Gao, L., Din, Z. U., Kim, K., Senouci, A., Han, Z., and Zhang, Y. (2025). Large Language Model-driven code compliance checking in Building Information Modeling. Electronics (Basel), 14(11):2146.

Nithyanantham, B. K., Sesterhenn, T., Nedungadi, A., Garijo, S. P., Zenkner, J., Bartelt, C., and Lüdtke, S. (2025). MCP4IFC: IFC-Based Building Design Using Large Language Models. arXiv:2511.05533 [cs].

Novikov, A., Vũ, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A. Z., Shirobokov, S., Kozlovskii, B., Ruiz, F. J. R., Mehrabian, A., Kumar, M. P., See, A., Chaudhuri, S., Holland, G., Davies, A., Nowozin, S., Kohli, P., and Balog, M. (2025). AlphaEvolve: A coding agent for scientific and algorithmic discovery. arXiv:2506.13131 [cs].

Shi, J. W. L., Solihin, W., and Yeoh, J. K. W. (2025). Fine-tuning a large language model for automated code compliance of building regulations. Adv. Eng. Inform., 68:103676.

Solibri Inc. (2026). Solibri Office.

Solihin, W. (2004). Lessons learned from experience of code-checking implementation in Singapore. In Building SMART Conference, Singapore.

Solihin, W. and Eastman, C. (2015). Classification of rules for automated BIM rule checking development. Autom. Constr., 53:69–82.

Wang, X., Chen, Y., Yuan, L., Zhang, Y., Li, Y., Peng, H., and Ji, H. (2024). Executable Code Actions Elicit Better LLM Agents. arXiv:2402.01030 [cs].

Wang, Z., Fuchs, S., Wu, J., Esser, S., Wrabel, T., and Borrmann, A. (2026). GNI BIM Dataset. Technical University of Munich, Georg Nemetschek Institute. Zenodo. `https://doi.org/10.5281/zenodo.19722012`.

Ying, H. and Sacks, R. (2024). From Automatic to Autonomous: a Large Language Model-driven Approach for Generic Building Compliance Checking. In Proceedings of the 41st International Conference of CIB W78.

Zhang, J. and El-Gohary, N. M. (2017). Integrating semantic NLP and logic reasoning into a unified system for fully-automated code checking. Autom. Constr., 73:45–57.

Zheng, J. and Fischer, M. (2023). Dynamic prompt-based virtual assistant framework for BIM information search. Autom. Constr., 155:105067.

Zheng, Z., Han, J., Chen, K.-Y., Cao, X.-Y., Lu, X.-Z., and Lin, J.-R. (2026). Translating regulatory clauses into executable codes for building design checking via large language model driven function matching and composing. Eng. Appl. Artif. Intell., 163(112823):112823.