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

AN EXTENSIBLE GRAPHQL API FOR FINE-GRAINED ACCESS TO BUILDING INFORMATION MODELS

Nepomuk Wolf[1,2], Sebastian Esser[1,2], and André Borrmann[1,2]
[1] Chair of Computing in Civil and Building Engineering, Technical University of Munich, Germany
[2] TUM Georg Nemetschek Institute, Technical University of Munich, Munich, Germany

Abstract

Building information exchange in large projects remains largely file-based, although many applications require only small, task-specific subsets of data. This paper addresses the need for element-level, transactional access to building information in web-based BIM workflows, including visualization and model analysis. While direct IFC-to-web mappings are technically feasible, they often result in complex schemas and verbose queries. We present a simplified GraphQL API that enables fine-grained access to building information while abstracting IFC complexity. Through atomic geometry resources, flattened property access, and a modular schema structure, the approach improves usability, query readability, and application-oriented BIM data access.

Introduction

Efficient information retrieval is a fundamental requirement of modern Building Information Modeling (BIM) workflows. While the Industry Foundation Classes (IFC) standard provides a comprehensive framework for representing geometry, topology, and rich semantic information, current practice remains largely centered around file-based exchange. IFC models are typically transferred as monolithic STEP files, forcing client applications to load entire datasets, even when only a small subset of information is required. This paradigm creates information silos that hinder low-latency data access and complicate the integration of emerging workflows, such as web-based visualization, digital twins, and AI-assisted analysis.

As the Architecture, Engineering, Construction and Operation (AECO) industry moves toward higher levels of BIM maturity, the demand for element-level, transactional access to model data is increasing. Many practical use cases require only partial model views, such as web-based visualization, quantity takeoff, or coordination workflows. In these scenarios, transferring and processing complete IFC files introduces unnecessary overhead and latency.

Previous work by Wolf et al. (2025) demonstrated that a direct 1:1 mapping between the IFC schema and GraphQL is technically feasible. However, the resulting API closely mirrored the complexity of the IFC schema, so even simple queries required traversing deeply nested relationships such as property assignment or geometric representations. Moreover, the heterogeneity of IFC geometry forces clients to possess detailed prior knowledge of internal modeling constructs.

This observation parallels a broader development in the Linked Data ecosystem of the building industry, where direct schema mappings have increasingly been replaced by more abstract and modular representations of building information.

Inspired by this paradigm shift, the research presented in this paper moves away from strict schema mirroring and toward the design of a tailored GraphQL API for BIM data access. Instead of exposing the full internal structure of IFC, the proposed interface provides a lightweight, extensible skeleton that prioritizes simplicity and practical usability. Complex hierarchical constructs are flattened, geometry and properties are treated as atomic leaf nodes, and building elements are exposed as directly queryable resources. At the same time, the API remains agnostic to the underlying data source and can operate on traditional IFC STEP files as well as with alternative storage backends.

In contrast to Large Language Model (LLM)-based interfaces, which may support natural-language interaction but do not themselves guarantee deterministic or structurally validated access to BIM data, GraphQL provides a strongly typed and self-documenting interface with automatic query validation.

The research question of this paper is: How can a tailored GraphQL API support fine-grained, interoperable access to building information while reducing query complexity compared to direct schema mappings? The scope of this work is read-oriented access for AECO workflows, with a particular focus on spatial structure, properties, and geometry.

This paper presents the architecture and design decisions of this simplified GraphQL interface, whose public software repository is cited in (Wolf, 2026), and shows how modular schema extensions enable the integration of external data sources, such as data sheets, databases, or sensor platforms, without requiring disruptive API versioning.

Technical Background and Related Works

GraphQL is a web-based API paradigm that combines a schema definition language, a query language, and a runtime for resolving client-defined queries. Originally introduced to address limitations of REST-based data fetching, such as over- and under-fetching of resources, it has since become a widely adopted approach for structured

data access. GraphQL APIs are strongly typed and self-documenting, allowing client applications to precisely define the structure and scope of requested responses. These characteristics make GraphQL particularly relevant for BIM-related scenarios in which clients frequently require only a small, task-specific subset of a much larger building model.

In recent years, GraphQL has also attracted growing interest within the AECO research community, where its potential for structured and fine-grained access to BIM-related data has been explored (Alexiev et al., 2023; Werbrouck et al., 2019b,a). Beyond the BIM domain, studies have evaluated GraphQL as an alternative to REST-based APIs. While REST can achieve lower raw response times in static scenarios, GraphQL has been shown to use resources more efficiently when data requirements change frequently (Lawi et al., 2021). Its declarative query model also mitigates overfetching, which is particularly relevant for BIM applications with large models (Quiña-Mera et al., 2023).

From Monolithic IFC Mappings to Modular Semantic Representations

The move toward tailored, API-driven access to IFC data is part of a broader evolution in building data standards, which increasingly favor modular, element-level information retrieval over monolithic, file-based exchange (Zhu et al., 2025). Early attempts to integrate IFC with web technologies were led by the Linked Data community. The ifcOWL ontology provides a 1:1 bidirectional mapping of the EXPRESS-based IFC schema into RDF (Beetz et al., 2009). While semantically complete, this approach resulted in very large graphs and highly verbose query patterns, mainly due to ordered collections and objectified relationships (Pauwels et al., 2015). As a response, the Linked Building Data community shifted toward smaller modular ontologies such as the Building Topology Ontology (BOT) (Rasmussen et al., 2021) and the Building Element Ontology (Pauwels, 2018). This transition demonstrated the benefits of lightweight structural skeletons that preserve essential semantics while reducing query and modeling complexity.

GraphQL and Semantic Web Approaches for Building Data Access

Rather than treating GraphQL and Semantic Web technologies as competing paradigms, recent research has explored their integration. Several approaches propose GraphQL as a client-facing interface for RDF and SPARQL backends (Gleim et al., 2020; Karalis et al., 2023). Taelman et al. (2019) compare architectural patterns for bridging GraphQL and RDF, while Angele et al. (2022) demonstrate that GraphQL can reduce query complexity for RDF-based datasets and enable closer alignment with ontology-driven data models.

A similar challenge arises when exposing IFC data through GraphQL. Wolf et al. (2025) demonstrated that a direct 1:1 mapping from EXPRESS to GraphQL is technically feasible. However, the resulting schema inherited much of the structural complexity of IFC, leading to deeply nested queries and frequent reliance on inline fragments. In addition, GraphQL lacks native support for several EXPRESS features, which further complicates schema design and client-side querying. This limitation mirrors earlier experiences from the Linked Data community: preserving the entire source schema maximizes formal completeness, but often at the cost of practical usability.

Toward Modular and Transactional BIM Data Exchange

In parallel to these developments, the IFC 5 roadmap explicitly promotes modularization, normalization, and object-level data exchange to support transactional workflows and partial model access (van Berlo et al., 2021). This direction aligns closely with the design motivation of the present work.

Geometry representation poses a comparable challenge in both GraphQL- and RDF-based approaches. In ifcOWL, geometry is mapped directly according to the IFC schema, which leads to extremely large datasets and high structural complexity (Pauwels et al., 2017). Wagner et al. (2020) provide an overview of alternative geometry handling strategies in Linked Data. Bonduel et al. (2019) demonstrate the possibility of abstracting geometry into atomic entities and embedding external formats such as OBJ or glTF. These findings align with the design decisions presented in this work, which treat geometry as an atomic resource optimized for downstream visualization and analysis tasks.

Proposed GraphQL API Design

The proposed API design follows four primary objectives: (1) to enable fine-grained, element-level access to building information without requiring full-file transfers, (2) to reduce query complexity by abstracting verbose IFC relationship structures, (3) to remain agnostic to the underlying data representation in order to support heterogeneous backends, and (4) to provide a modular foundation that can be extended with domain-specific information without introducing breaking changes. These objectives guide the simplification of spatial structures, the flattening of property access, the abstraction of geometry representation, and the modular extension mechanism described in the following sections. Rather than adopting a strict 1:1 mapping of the IFC schema, the proposed design follows a handcrafted, lightweight core schema that captures only the essential structural concepts. This should be understood as a task-oriented abstraction rather than a complete reproduction of the original data model. Although the initial development was driven by IFC models and their STEP-based serialization, the resulting API is not tied to a specific source format. Instead, the modular architecture enables the resolution of data from heterogeneous backends, such as files in arbitrary formats (for example RDF, JSON,

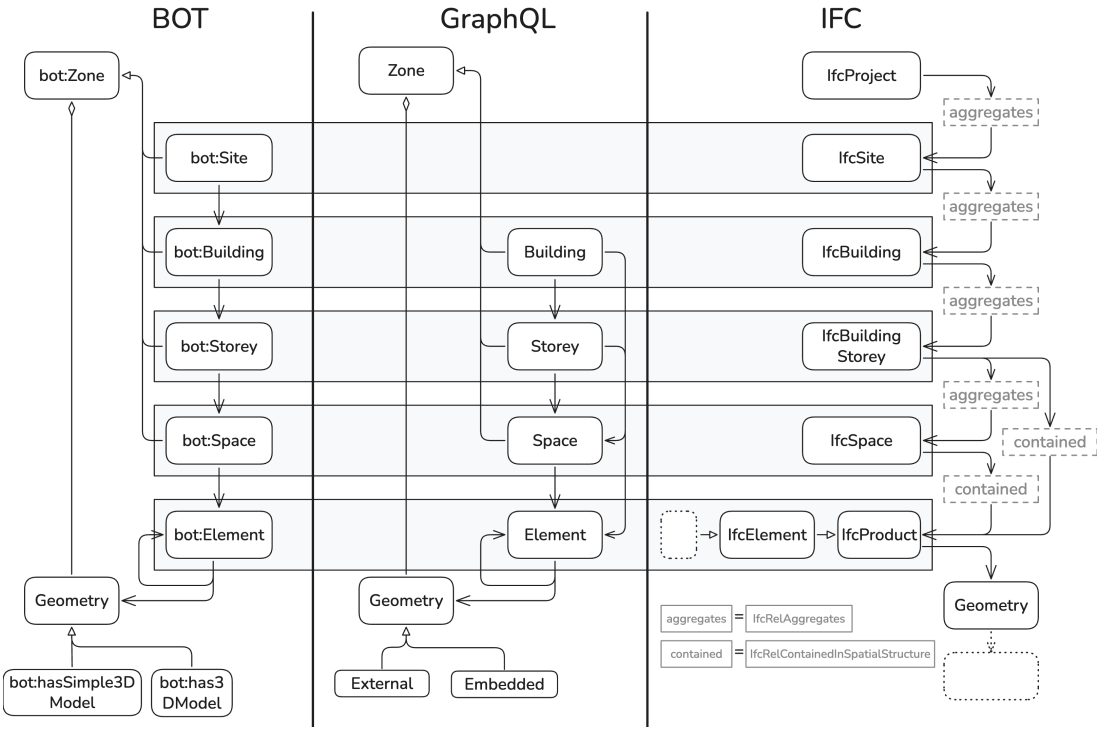

Figure 1: Similarities and differences between the spatial structure in BOT, IFC, and the GraphQL API

XML) or databases.

Spatial Structure and Topology

The core of the API design is a simplified spatial hierarchy that is conceptually aligned with the Building Topology Ontology (BOT) while remaining compatible with common IFC modeling practices. This structure provides a simplified navigation model for client applications and serves as the primary entry point for accessing spatially organized building data. To reduce traversal complexity, the API adopts a hierarchical zone model based on BOT and introduces a common `Zone` interface implemented by the `Building`, `Storey`, and `Space` types. In contrast to IFC, where spatial containment is expressed through objectified relationships (e.g., `IfcRelAggregates`), these relationships are represented directly as fields of the respective types. This significantly reduces query depth and improves readability while preserving the underlying semantic structure. Figure 1 illustrates the resulting schema structure and highlights the similarities and differences between BOT, IFC, and the proposed GraphQL API. Despite substantial simplification, the high-level organization remains closely aligned with IFC's spatial concepts. The most notable deviation is the elimination of intermediary relationship entities, which are replaced by direct associations exposed at the API level.

To support spatial reasoning and topological queries, the API additionally exposes explicit relationship fields between zones, including `partOf`, `intersects`, and `adjacent`. These relations follow the conventions established by BOT and enable expressive spatial queries without requiring clients to traverse low-level relational constructs. Building elements such as walls, windows, and slabs are represented using a generic `BuildingElement` type instead of directly mirroring the deep inheritance hierarchy of IFC. The concrete IFC subtype is preserved as a field on the `BuildingElement` type. This design trades explicit schema-level inheritance for a flatter and more compact API structure while trying to maintaining access to semantic type information. Element-level containment is supported through the `contains` and `partOf` fields, enabling the representation of nested component relationships, such as windows hosted within walls. In addition, building elements can be attached to any of the `Zone` subclasses (Building, Storey, Space), following the flexible attachment model defined in BOT. This differs from typical IFC usage patterns, where most building elements are associated with `IfcBuildingStorey` and furnishing elements with `IfcSpace`. Listing 1 shows an excerpt of the corresponding GraphQL schema definition.

Not all semantically rich IFC structures are directly exposed in the compact core schema. More complex concepts, such as material assignments or links to external classification systems, can instead be resolved dynamically through dedicated resolver functions. In this way, the simplified API surface remains stable, while the server can still retrieve and integrate richer information from IFC relationships or from additional backend systems when required.

Flattened Property Access

A direct 1:1 mapping of the IFC property hierarchy results in verbose queries that require traversing multiple intermediate entities such as `IfcPropertySet` and

```
interface Zone {
    id: ID!
    name: String
    geometry: Geometry
    contains: [Zone]
    elements: [BuildingElement]
    partOf:[Zone]
    intersects: [Zone]
    adjacent: [Zone]
}

type Building implements Zone{...}

type Storey implements Zone{...}

type Space implements Zone{...}
```

Listing 1: GraphQL schema excerpt for spatial structure and zone hierarchy

`IfcPropertySingleValue`. While this level of detail is valuable for authoring and editing workflows, it provides limited benefit for most read-oriented client applications and significantly reduces query readability.

To address this issue, the proposed API collapses objectified property relationships and exposes properties directly as a field of the `BuildingElement` type.

In IFC, properties are additionally organized into property sets (psets), which provide semantic grouping and contextual meaning. Preserving this information while maintaining a flattened access pattern can be achieved in two ways. The first approach is to explicitly model property sets as separate entities attached to each building element, each containing a collection of properties. The second approach is to associate a `pset` attribute directly with each property, thereby retaining grouping information without introducing an additional layer of nesting.

Both approaches preserve the original semantics, however, they differ in terms of query ergonomics rather than expressive power. Since typical client queries focus on retrieving individual properties rather than entire property sets, the second approach is adopted. This design minimizes structural overhead while still allowing semantic grouping information to be accessed when required.

Listing 2 shows that the association between properties and property sets is preserved in two complementary ways. First, the `pset` field can be explicitly requested as part of the `Property` object. Second, the `properties` field of the `BuildingElement` type supports filtering by property set name, enabling selective retrieval of properties belonging to a specific `pset`.

Atomic Geometry Representation

Geometry is one of the most relevant information categories in building models. It also represents one of the most complex aspects of the IFC data structure. A direct 1:1 mapping of IFC geometry entities into GraphQL needs to represent highly verbose and heterogeneous geometry representations, including boundary representations (BREP), swept solids, meshes, constructive solid geometry (CSG), and parametric extrusions. As a result, client applications either need prior knowledge of the underlying modeling paradigm or need to request multiple alternative representations simultaneously, in order to formulate valid queries.

```
type BuildingElement{
    geometry: Geometry
    guid: ID!
    name: String
    contains: [BuildingElement]
    partOf: [BuildingElement]
    properties(pset: String): [Property]
}

type Property {
    name: String
    value: JSON
    pset: String
}
```

Listing 2: GraphQL schema excerpt for building elements

This limitation conflicts with the declarative nature of GraphQL, where clients explicitly specify the expected response structure. Since the geometric representation type of an element is not known in advance, a fine-grained schema-level exposure of IFC geometry significantly increases query complexity.

To address this issue, geometry is treated as an atomic attribute within the API design. In this context, atomic means that the client accesses the geometry as a single leaf node rather than traversing the internal construction logic of the IFC model. Instead of exposing the internal structure, the complete geometry of an element is abstracted behind a single `geometry` field and can therefore be returned as a single unit in the GraphQL response. This design removes the need for the client to know in advance whether the underlying IFC geometry was modeled as BREP, extrusion, CSG, or another representation type.

This abstraction is intentionally lossy. Part of the expressive power of the original IFC representation is no longer directly available once geometry is exported into a downstream exchange format or otherwise reduced to an atomic resource. In particular, procedural and parametric geometry definitions are typically not preserved when exporting to commonly used geometry exchange formats, which limits the ability to reconstruct design intent or perform fine-grained semantic reasoning on geometric constructs. The same tradeoff applies more generally throughout the proposed schema design: direct access to the full internal IFC graph is reduced in favor of improved query readability and simpler client-side interaction.

```
"guid": "0Czv28JAHuep4",
"geometry": {
  "external": {
    "uri": "https://.../0Czv28JAHuep4.obj",
    "extension": "obj"
  },
  "embedded": {
    "payload": "
    v 1.22342604 2.82471817 0.00000000
    v 1.22342604 2.82471817 3.00000000
    v 0.54726129 0.30123695 3.00000000
    ...
    ",
    "encoding": "none",
    "extension": "obj"
  }
}
```

Listing 3: Response snippet with both external and embedded geometry

Geometry Transfer Strategies

The API supports two complementary strategies for transferring geometric data, depending on performance requirements and application context.

External Geometry: In this approach, geometry is stored in an external resource such as a file server or object storage system. The corresponding resolver returns a URI referencing the geometry file. This strategy enables efficient handling of large datasets, supports streaming and caching, and avoids inflating GraphQL response payloads.

Embedded Geometry: Alternatively, geometry can be embedded directly into the GraphQL response. Since GraphQL responses are (typically) encoded as JSON, the geometric representation must be serializable into a compatible format. This can be achieved in two ways:

- Text-based geometry formats, such as OBJ or WKT can be embedded directly as strings.
- Binary geometry formats can be encoded as a string (for example using Base64-encoding) and transmitted together with metadata describing the format and encoding method.

While Base64 encoding increases payload size and requires decoding on the client side, it enables the transfer of arbitrary geometry formats in a self-contained manner. For this reason, both embedded and externally referenced geometry representations are supported by the prototype API.

To demonstrate the flexibility of this abstraction, the prototype implementation supports multiple geometry formats, including OBJ, GLTF, GLB, WKT, BREP, and STL. The schema itself remains agnostic to specific formats and can be extended without modifying the core API definition. Each client application has full control over the format of the returned geometry and can choose between one of the supported formats. Listing 3 shows a response object, containing geometry in the OBJ format both as external and embedded representation.

Modular Schema Design

A central design principle adopted from the Linked Building Data community is to keep the core schema compact while enabling domain-specific extensions. This separation allows the API to remain lightweight and stable, while still supporting specialized application requirements such as energy analysis, cost estimation, or environmental assessment. GraphQL natively supports this paradigm, as types and fields can be extended without introducing breaking changes or requiring explicit API versioning.

Unlike REST-based interfaces, where modifications to response payloads often require version management to avoid disrupting existing clients, GraphQL enables clients to explicitly define the requested response structure. Newly added fields therefore remain invisible to existing client applications unless they are explicitly queried. This property enables incremental schema evolution while preserving backward compatibility.

Beyond the static core schema, the proposed implementation introduces a modular extension mechanism that dynamically loads schema fragments together with their associated resolver functions. Each extension contributes additional fields and data sources without modifying the base API definition. Resolver functions establish the connection between these abstract schema fields and concrete backend services and are executed only when the corresponding fields are requested by the client.

To demonstrate this mechanism, an example extension was implemented that augments the `BuildingElement` type with a `dataSheetURL` attribute. This field establishes a link between a building element and an external product data sheet, when such information is available. The referenced data is not part of the base API schema and is typically not contained in IFC models, yet it is highly relevant for many downstream workflows such as specification, procurement, and facility management.

Listing 4 illustrates the schema snippet required to extend the core API. In addition to the schema extension, a corresponding resolver function is provided that dynamically searches for and retrieves a matching product data sheet URL when the field is requested by a client. The same mechanism can also be used to expose more complex information that is not represented as a simple core field, such as material associations or external classification data. This example demonstrates how heterogeneous external information sources can be integrated transparently into the API without modifying the underlying core data model.

Evaluation and Discussion

This section presents a qualitative evaluation of the proposed API design with respect to query expressiveness, modular extensibility, and practical limitations. Rather

```
extend type BuildingElement {
    dataSheetURL: String
}
```

Listing 4: GraphQL schema extension for linking external product data sheets

than benchmarking response times or comparing implementation performance, the evaluation focuses on the usability of the schema through representative query scenarios and the prototypical API implementation. The selected examples were chosen to cover two central requirements of purpose-driven BIM data access: retrieving connected information across multiple relationship levels and integrating external information without modifying the core schema.

Query Example 1: Connected Data Retrieval

Many BIM workflows require access not only to individual elements but also to related entities connected through spatial, containment, or topological relationships. Typical examples include retrieving all components of a storey together with their subcomponents, or navigating from structural elements to embedded openings or installations. In conventional REST-based APIs, such workflows would require multiple sequential requests and additional client-side aggregation logic. The proposed GraphQL API enables retrieval of connected information across multiple relationship levels within a single query. Figure 2 illustrates this capability by querying all load-bearing walls of a given storey and the windows contained within each wall. The query traverses the spatial hierarchy and containment relations and explicitly specifies the required response structure, in this case *guid* and *name* for each wall and *geometry* for each window. This example was selected because it combines several of the central design goals of the proposed schema: element-level filtering, traversal across hierarchical and containment relationships, and access to geometry as an atomic resource. It therefore demonstrates how the API supports connected data retrieval in a compact and application-oriented way. This example also showcases the integration of geometry in the form of a reference to an external resource.

Query Example 2: Modular Data Enrichment via Schema Extensions

Many analysis and decision-support workflows require information that is not natively contained in IFC models, such as environmental indicators, cost metrics, or product-specific metadata. As described previously, the proposed API addresses this challenge through a modular extension mechanism that allows additional fields to be introduced.

Figure 3 shows the result containing information from the IFC model alongside additional information in the form of the reference to an external data sheet related to the element. This example was chosen to evaluate extensibility rather than traversal complexity. It demonstrates that the API can federate information from the core building model together with external resources through resolver-based schema extensions, without requiring changes to existing query structures or disruptive API versioning.

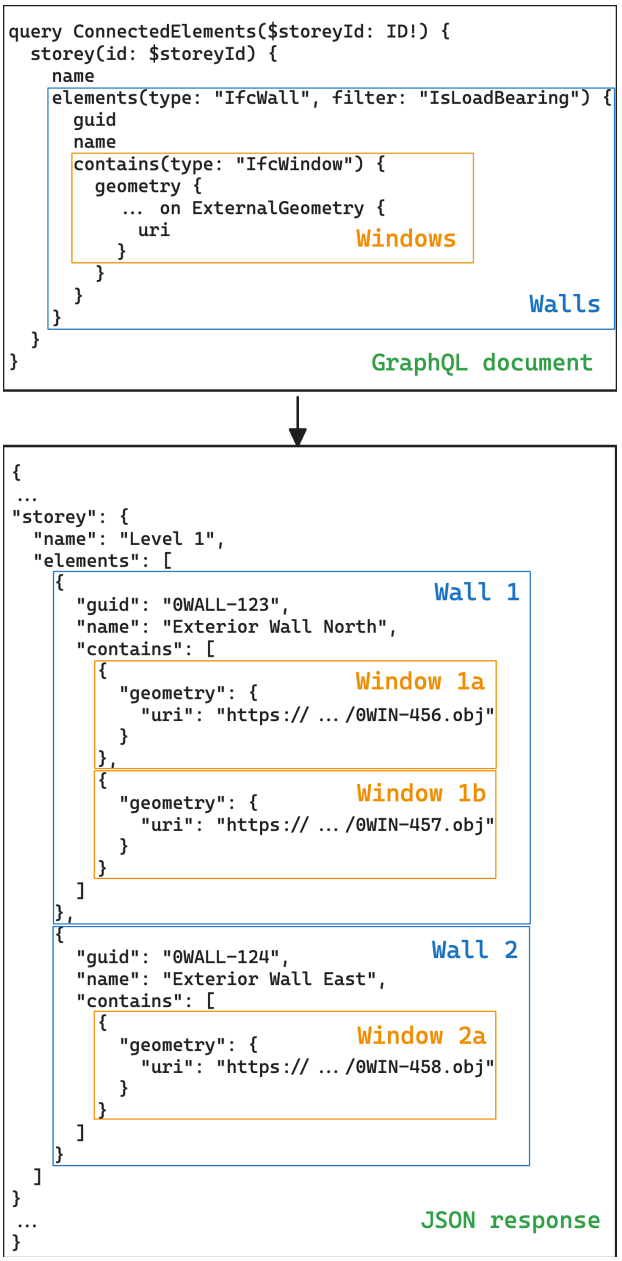

Figure 2: Requesting related elements with filtering

A key conceptual difference between the proposed GraphQL interface and SPARQL-based access to building data lies in the interaction model and query abstraction level. SPARQL provides direct access to RDF graphs and exposes the full underlying data model, which enables complex pattern matching but requires detailed knowledge of ontology structures. In contrast, the proposed API deliberately abstracts underlying data representations and exposes a task-oriented schema that is tailored to common BIM access patterns. The GraphQL interface encapsulates

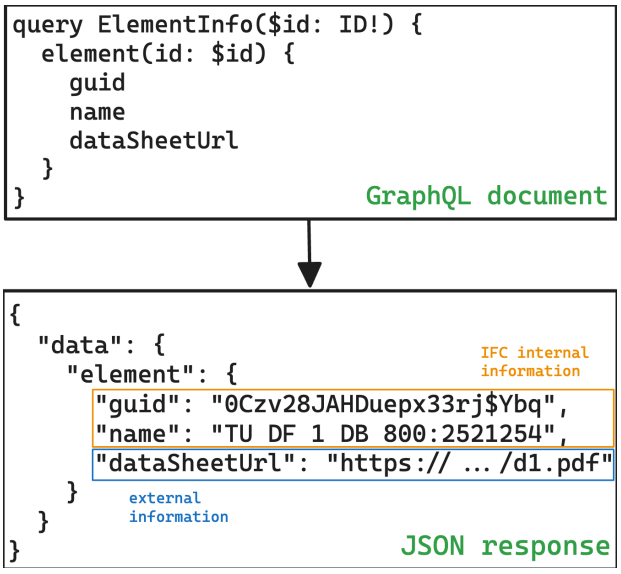

Figure 3: Query containing external information

complex graph traversal logic within resolver implementations, shifting complexity from the client to the server. At the same time, this abstraction comes at the cost of reduced direct access to the full underlying graph structure.

Scope and Limitations

The proposed API intentionally departs from a strict 1:1 mapping of the IFC schema in favor of a simplified data model. While this improves usability and query readability, it also implies that not all IFC semantics and modeling constructs are directly exposed at the API level.

The abstraction of geometry into atomic representations introduces additional limitations. Mesh-based exchange formats commonly used for visualization and data transfer, such as OBJ or glTF, do not preserve the full semantic and parametric structure of the original IFC geometry. Once converted, it is no longer possible to determine whether a shape was originally defined as BREP, CSG, or procedural description. It could be explored if extracting the geometry as a valid subset from the IFC model could be an alternative to typical geometry exchange formats.

Furthermore, certain geometric constructs cannot be represented exactly using mesh-based formats and require approximation. While these representations are sufficient for most visualization and downstream analysis tasks, they may lead to information loss in workflows that depend on higher-level modeling semantics. A detailed assessment of these trade-offs is provided in Wagner et al. (2020).

Several aspects did not fit the scope of this work and should be explored in future works.

First, the current implementation focuses on read-oriented access patterns. Future development should include support for GraphQL mutations to enable write operations, such as updating or creating (new) elements.

Second, the modular extension mechanism could be expanded to cover additional domain-specific use cases, for example quantity take-off, cost estimation, energy performance indicators, scheduling information, and sensor data integration.

Third, although the API design is resource-agnostic in principle, the current prototype only resolves data from IFC STEP files. Future work should extend the resolver implementations to support alternative IFC representations, such as graph databases, as well as native Linked Data representations.

Finally, this work does not address authentication, authorization, and access control mechanisms. Incorporating these aspects is essential for real-world deployment and represents an important direction for future research.

Conclusion

This paper presented a GraphQL-based API design for purpose-driven, fine-grained access to IFC building models. By moving away from a strict one-to-one mapping of the EXPRESS schema and instead adopting a flattened, modular interface inspired by Linked Building Data principles, the proposed approach reduces query complexity while supporting element-level, transactional access for read-oriented AECO workflows.

The presented schema design addresses two central challenges of IFC-based web access: verbose geometry retrieval and deeply nested property hierarchies. Treating geometry as an atomic, format-agnostic resource enables flexible integration of external and embedded representations, while flattened property access improves usability for common analysis and visualization workflows. In addition, the modular extension mechanism demonstrates how domain-specific information can be integrated dynamically without breaking existing clients or requiring API versioning.

The qualitative evaluation illustrates how GraphQL enables traversal of connected building information within a single request and supports the integration of external information through schema extensions. At the same time, the proposed design intentionally trades direct access to the full internal IFC structure for improved query readability and simpler client-side interaction. The resolver-based architecture further decouples the API schema from underlying data sources.

Overall, this work shows that GraphQL provides a practical and structurally explicit foundation for application-oriented access to building information models.

Acknowledgement

The research presented has been financially supported by the Federal Ministry of Education and Research (Bundesministeriums für Bildung und Forschung) in the frame of the project BauPuls360 under grant number 02K23A096, and by the German Research Foundation (DFG) in the frame of the Research Unit FOR 5672 (grant number 517965147). The responsibility for the content of this publication lies with the authors.

During the preparation of this paper, the authors used ChatGPT 5.3 to assist with writing and language refinement. All intellectual content, analysis, and conclusions remain the authors' responsibility.

References

Alexiev, V., Radkov, M., and Keberle, N. (2023). Semantic bSDD: improving the GraphQL, JSON and RDF representations of buildingSMART data dictionary. In Proc. of the LDAC Conference, pages 85–97.

Angele, K., Meitinger, M., Bußjäger, M., Föhl, S., and Fensel, A. (2022). GraphSPARQL: a graphql interface for linked data. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, pages 778–785.

Beetz, J., van Leeuwen, J., and Vries, d. (2009). IfcOWL: A case of transforming EXPRESS schemas into ontologies. AI EDAM, 23:89–101.

Bonduel, M., Wagner, A., Pauwels, P., Vergauwen, M., and Klein, R. (2019). Including widespread geometry formats in semantic graphs using RDF literals. In EC3 Conference 2019, volume 1, pages 341–350. European Council on Computing in Construction.

Gleim, L. C., Holzheim, T., Koren, I., and Decker, S. J. (2020). Automatic bootstrapping of graphql endpoints for rdf triple stores. In ASLD 2020: Advances in Semantics and Linked Data — Joint Workshop Proceedings from ISWC 2020, volume 2722 of CEUR Workshop Proceedings, pages 119–134, Aachen, Germany. CEUR-WS.org.

Karalis, N., Bigerl, A., and Ngonga Ngomo, A.-C. (2023). Native Execution of GraphQL Queries over RDF Graphs Using Multi-Way Joins. In Knowledge Graphs: Semantics, Machine Learning, and Languages. IOS Press.

Lawi, A., Panggabean, B. L. E., and Yoshida, T. (2021). Evaluating GraphQL and REST API Services Performance in a Massive and Intensive Accessible Information System. Computers, 10(11):138.

Pauwels, P. (2018). Building element ontology. `https://pi.pauwel.be/voc/buildingelement/`, accessed 2026-04-03.

Pauwels, P., Krijnen, T., Terkaj, W., and Beetz, J. (2017). Enhancing the ifcOWL ontology with an alternative representation for geometric data. Automation in Construction, 80:77–94.

Pauwels, P., Terkaj, W., Krijnen, T., and Beetz, J. (2015). Coping with lists in the ifcOWL ontology. In Proceedings of the 22nd EG-ICE Workshop 2015, 13-15 June 2015, Eindhoven, The Netherlands, pages 111–120.

Quiña-Mera, A., Fernandez, P., García, J. M., and Ruiz-Cortés, A. (2023). GraphQL: A Systematic Mapping Study. ACM Comput. Surv., 55(10):202:1–202:35.

Rasmussen, M. H., Lefrançois, M., Schneider, G. F., and Pauwels, P. (2021). BOT: The building topology ontology of the W3C linked building data group. Semantic Web, 12(1):143–161.

Taelman, R., Vander Sande, M., and Verborgh, R. (2019). Bridges between GraphQL and RDF. In W3C Workshop on Web Standardization for Graph Data. W3C, pages 4–7.

van Berlo, L., Krijnen, T., Tauscher, H., Liebich, T., van Kranenburg, A., and Paasiala, P. (2021). Future of the Industry Foundation Classes: towards IFC 5. In Proc. of the 38th International Conference of CIB W, volume 78, pages 11–15.

Wagner, A., Bonduel, M., Pauwels, P., and Rüppel, U. (2020). Representing construction-related geometry in a semantic web context: A review of approaches. Automation in Construction, 115:103130.

Werbrouck, J., Senthilvel, M., Beetz, J., Bourreau, P., and Van Berlo, L. (2019a). Semantic query languages for knowledge-based web services in a construction context. In 26th International Workshop on Intelligent Computing in Engineering, EG-ICE 2019, volume 2394, page 3.

Werbrouck, J., Senthilvel, M., Beetz, J., and Pauwels, P. (2019b). Querying heterogeneous linked building data with context-expanded GraphQL queries. In 7th International Workshop on Linked Data in Architecture and Construction, volume 2389 of CEUR Workshop Proceedings, pages 21–34. CEUR-WS. org.

Wolf, N. (2026). graphql-building-api. Software repository. URL: `https://github.com/NepomukWolf/graphql-building-api`.

Wolf, N., Esser, S., and Borrmann, A. (2025). IFC2GraphQL: schema mapping and automated api construction from ifc to graphql. In Proceedings of the 42nd International CIB W78 Conference.

Zhu, J., Nisbet, N., Yin, M., Wei, R., and Brilakis, I. (2025). Releasing the power of graph for building information discovery. Automation in Construction, 172:106034.