Towards Conversational BIM Agents: Generate 3-D BIM Blocks in Revit Using an LLM

Joseph Mathew Aerathu, Sharmista Debnath, Patrick Kastner

Georgia Institute of Technology, Atlanta, USA

Abstract

Recent advancements in LLMs have shown promise in translating natural language into structured digital assets, yet their application to 3D-modeling, especially BIM, which appends "information" and data to models, remains in its early stages. This paper presents an open-source, end-to-end pipeline that converts natural language architectural prompts into OBJ mesh geometry, and then parsing and cleaning it to generate IFC models viewable in BIM platforms, such as Autodesk Revit. The workflow utilizes openAI's GPT-4o-mini model, which is one of the newest and most efficient GPT models with reasoning capabilities, accessed via the openAI API. It is coupled with a lightweight OBJ mesh parser and the IfcOpenShell API. First, the LLM generates an OBJ representation of the requested geometry; second, a parser extracts vertices and faces, accommodating both triangular and quad elements; finally, the mesh is wrapped as an IfcFacetedBrep within a minimal IFC hierarchy and exported as a generated .ifc block model. The modular three-script design enables model-agnostic swapping of either the language model or the geometry converter, supporting extensibility to alternative formats or locally hosted models. An MVP demonstrates successful round-trip generation and import of basic building masses in Revit 2023 under the IFC 2×3 schema, with OpenAI's GPT-4o-mini model outperforming a fine-tuned LLaMA-Mesh baseline in mesh fidelity. Current limitations include reliance on the LLM's geometric accuracy, absence of semantic class assignment beyond IfcBuildingElementProxy, and lack of unit handling or mesh validation. Ongoing work targets enriched IFC metadata, multi-object prompts, eventual direct .RVT file generation, and developing a model capable of extending its reasoning capabilities to 3D-modeling. By bridging natural-language design intent and standards-compliant BIM output, this Text-to-IFC framework lowers the barrier to rapid prototyping and lays the groundwork for conversational BIM copilots, as well as, future agentic 3d-modeling capabilities/environments.

Key Innovations

- Reusable pipeline swap-friendly between different chat models; you can swap GPT 4o-mini for another chat model.
- Extensible parser/converter architecture for diverse file formats; so you can replace either part, incase you want to implement this for another file format.
- Minimal IFC hierarchy generation with IfcFacetedBrep.
- Real-time building element proxy creation for immediate Revit import.
- Open-source; all the work can be accessed at this github repo: https://github.com/jma1999/ARCH-8833-Sp25-LLM2IFC/tree/main.

Practical Implications

This workflow empowers architects and engineers to automate BIM 3d-model geometry generation, reducing manual modeling errors and accelerating design iteration cycles.This also paves the way to making 3d-modeling accessible to everyone. LLMs have made coding accessible to everyone; 3d-mesh geometry is essentially just a text document of code describing vertices and faces. Combining these functionalities, we make 3d-modeling accessible and reduce the barrier to entry to modeling information along with the model. 3d-modeling using LLMs also paves the way to applications beyond BIM within the AEC sector, like BEM, as well as applications beyond AEC, in other sectors that implement digital twins with data.

Introduction

Manual OBJ-to-IFC conversion is time-intensive and error-prone. Recent LLMs demonstrate text-to-code proficiency but lack direct BIM integration. We address this by asking: *Can we develop an end-to-end pipeline that automates BIM model generation from natural language?*

The integration of Natural Language Processing (NLP) into Building Information Modeling (BIM) workflows presents major advantages for the Architecture, Engineering, and Construction (AEC) industry by enhancing software usability, minimizing cognitive load, and improving interoperability. With over 1,000 commands, modern BIM programs like Revit and Vectorworks need users to navigate complex interfaces requiring significant learning and exertion (Du, Deng, et al., n.d.). NLP-driven command prediction systems demonstrates potential in streamlining user interactions, attaining 78.10% accuracy in predicting following actions; however, major challenges still remain in translating design intent across platforms and addressing data interoperability issues (Du, Deng, et al., n.d.; Nousias, n.d.). Up to 20–30% of data is compromised during IFC-to-Revit conversions, and proprietary BIM formats frequently result in data loss during system transitions (Jang et al., 2024). Automatic compliance validation is further hindered by semantic gaps between natural language regulations and structured BIM schemas, making it necessary for dynamic NLP techniques like transformer-based models refined on domain-specific corpora to increase concept alignment accuracy by 15% (Du, Nousias, et al., n.d.; Jang et al., 2024; Zheng & Fischer, n.d.).

One of the primary obstacles to adoption of BIM software is its complexity, which requires users to become proficient in hundreds of commands across disciplines, including structural engineering, architectural design, and MEP systems (Ashrafi, 2022; Du, Deng, et al., n.d.). With the use of transformer topologies from large language models (LLMs), sequential recommendation systems have demonstrated the ability to predic the next-best commands with 84% recall@10 accuracy, significantly reducing down on the navigation time in programs such as Vectorworks (Du et al., 2025; Rane et al., 2023). These systems characterize user interaction patterns by preprocessing massive amounts of BIM log data, allowing for real-time recommendations that align with project-specific workflows. The Dynamic Graph Neural Network for Sequential Recommendation (DGSR) framework, produces embeddings that capture structural and sequential information from past command sequences, resulting in strong performance in anticipating new user actions (Nousias, n.d.). However, current solutions overlook more significant issues since they only address command prediction rather than comprehensive workflow automation (Du, Deng, et al., n.d.; Du et al., 2025).

When importing models between open standards like Industry Foundation Classes (IFC) and proprietary formats like Revit (RVT), interoperability is still a major problem in BIM processes. The object-oriented system of Revit and the hierarchical structure of IFC collide, leading to misaligned MEP systems and broken parametric relationships during translation—wall assemblies lose material layers. While IFC to Brick Ontology semantic enrichment methods enhance facility management applications, they fail to address Revit compatibility, requiring teams to manually produce 20–30% of model data. Despite ignoring practical interoperability requirements, T5 transformers achieve 82% accuracy in abstract schema alignment, leaving crucial gaps in parametric family reconstruction (Jang et al., 2024; Li et al., 2024; Vo, n.d.). Even advanced frameworks, such as Text2BIM, which produces 99.4% accurate rule-compliant models, are limited in their usefulness for cross-platform collaborations because they require pre-existing templates rather than raw IFC inputs (Du et al., 2024; Du, Nousias, et al., n.d.).

The current NLP-BIM systems limit their use in dynamic design processes by giving information retrieval precedence over model update. Systems such as BIM-GPT allow for 81.9% accurate natural language queries of BIM data, however they are only passive helpers because they cannot change models (Du et al., 2025; Du, Nousias, et al., n.d.; Jang et al., 2024). Voice commands can update BIM elements using prototypes like DAVE, but they fail about half the time when compound instructions are used. This highlights technical difficulties in converting ambiguous natural language into exact API syntax (Fernandes et al., 2024). The potential for automation is undermined by multi-agent frameworks that divide work among specialized AI roles (such as programmers and architects) yet still call for human error-checking (Du, Nousias, et al., n.d.). The NADIA framework is an example of advancement in specific applications; with GPT-3.5, natural language prompts converted into JSON instructions for exterior walls that are compatible with Revit with 83.33% accuracy and 98.54% thermal standard compliance. However, NADIA's dependence on pre-processed Revit models emphasis on systemic limitations since without additional human intervention, no application can produce editable RVT files from IFC data directly (Jang et al., 2024).

The aim of this study is to examine the potential of developing a comprehensive pipeline that automates generating industry-standard IFC building models directly from natural language descriptions using Large Language Models (LLM). The proposed framework will be utilizing a chat based natural language interface to obtain the design inputs from the user. Preserving the spatial-parametric relationships, topological integrity and material properties in Revit environments during mesh to IFC conversion while accurately interpreting e architectural intent in LLMs are the key challenges. Integrating Revit, stem accuracy will validate how well the generated models match the brief description provided by the users in terms of geometric fidelity, component linkages and regulatory compliance criteria.

Methodology

In this section, we describe the end-to-end workflow that transforms a plain-language building description into a valid IFC block. The pipeline is composed of three independent scripts, each responsible for one major stage: mesh generation, parsing, and IFC construction.

Process Overview

Our six-step framework:

1. User prompt to LLM for OBJ-like mesh text.

2. Extract code block from LLM response.
3. Parse vertex/facet data into Python lists.
4. Wrap mesh as IfcFacetedBrep via IfcOpenShell.
5. Construct minimal spatial structure (Project, Site, Building, Storey).
6. Write IFC file and import into Revit.

Implementation Decisions

- **OBJ format**: simple grammar, well-supported syntax
- **Three-script split**: independent testing and swapping
- **IfcFacetedBrep**: universal viewer support
- **IfcOpenShell API**: high-level helper functions

System Architecture

The overall architecture comprises three Python scripts:

```
User prompt
    ↓
[01_prompt_llm.py]
    ↓
[02_parse_mesh.py]
    ↓
[03_mesh_to_ifc.py]
```

This modular design enables easy debugging, replacement, or extension of individual stages.

Stage 1: LLM-driven Mesh Generation

- **Prompt formulation:** A system message (“You are a helpful 3D-building assistant.”) and a user instruction (“Generate a small rectangular building block.”) are combined.
- **API call:** Using the OpenAI Python SDK (version $\geq$ 1.0), we call `client.chat.completions.create(model="gpt-4o-mini")` and save the full response to `obj_mess.txt`.
- **Rationale:** OBJ format’s simple “`v`” and “`f`” syntax is easy for LLMs to output and for us to parse.

Listing 1: Prompting script

```python
from openai import OpenAI

client = OpenAI(api_key="YOUR_OPENAI_API_KEY")
user_input = "Generate a simple small rectangular building block."
prompt_text = f"""
You are a building-model assistant.
Given user instructions about building geometry, output a 3D mesh in simplified OBJ text.
User instructions: "{user_input}"
Now produce an OBJ-like text with vertex (v) lines and face (f) lines.
"""
completion = client.chat.completions.create(
    model="gpt-4o-mini",
    store=True,
    messages=[
      {"role": "system", "content": "You are a helpful 3D-building assistant."},
      {"role": "user", "content": prompt_text}
    ]
)
print(completion.choices[0].message)
```

Stage 2: OBJ Extraction & Cleaning

- **Code-block isolation:** `extract_code_block()` scans for the first pair of triple backticks (“““) in `obj_mess.txt` and extracts only the enclosed lines.
- **Fallback:** If no fences are found, the entire file is treated as mesh text.
- **Debugging:** The extracted OBJ snippet is printed to the console for inspection.

Listing 2: Code block extractor

~~~python
def extract_code_block(full_text):
    """
    Return only the lines between the first pair of triple backticks (''').
    If no backticks found, fallback to entire text.
    """
    lines = full_text.splitlines()
    code_lines = []
    in_block = False
    for line in lines:
        if line.strip().startswith("```"):
            in_block = not in_block
            continue
        if in_block:
            code_lines.append(line)
    return "\n".join(code_lines) if code_lines else full_text
~~~

Listing 3: Mesh parsing

```python
def parse_obj_text(obj_str):
    vertices, faces = [], []
    for line in obj_str.splitlines():
        parts = line.strip().split()
        if not parts: continue
        if parts[0] == "v":
            x,y,z = map(float, parts[1:4])
            vertices.append((x,y,z))
        elif parts[0] == "f":
            indices = [int(p.split('/')[0]) for p in parts[1:]]
            faces.append(indices)
    return vertices, faces
```

Stage 3: Parsing Vertices and Faces

- Scan each line: lines starting with `v` yield `(x,y,z)` tuples; lines starting with `f` yield integer index lists (OBJ’s 1-based convention).
- Store in Python lists: `vertices: List<Tuple<float,float,float>>`, `faces:List<List<int>>`.
- Write `parsed_mesh.txt` summarizing all vertices and faces.

```python
with open("parsed_mesh.txt","w") as out:
    out.write("Vertices:\n")
    for v in vertices: out.write(f"{v[0]} {v[1]} {v[2]}\n")
    out.write("\nFaces:\n")
    for f in faces: out.write(" ".join(map(str,f))+"\n")
```

Stage 4: IFC Construction

1. Create a new IFC2X3 file with `ifcopenshell.file(schema="IFC2X3")`.
2. Define `IfcPerson`, `IfcOrganization`, `IfcOwnerHistory`, and SI units (`IfcSIUnit`).
3. Build spatial hierarchy: `IfcProject` → `IfcSite` → `IfcBuilding` → `IfcBuildingStorey`.

4. For each face:
   - Create `IfcCartesianPoint` per vertex.
   - Wrap points in `IfcPolyLoop`, `IfcFaceBound`, then `IfcFace`.
5. Aggregate faces into an `IfcClosedShell` & `IfcFacetedBrep`.
6. Encapsulate in `IfcShapeRepresentation` and `IfcBuildingElementProxy`, then write `GeneratedBlock.ifc`.

Listing 4: IFC conversion

```python
import ifcopenshell, ifcopenshell.guid

# Read parsed mesh
verts, faces = read_parsed_mesh_file("parsed_mesh.txt")

# Create IFC file
ifc = ifcopenshell.file(schema="IFC2X3")
# (OwnerHistory, Units, Context omitted for brevity)
# Build face entities
bounds = []
for face in faces:
    pts = [ifc.create_entity("IfcCartesianPoint", Coordinates=verts[i-1]) for i in face]
    loop = ifc.create_entity("IfcPolyLoop", Polygon=pts)
    fb = ifc.create_entity("IfcFaceBound", Bound=loop, Orientation=True)
    face_ent = ifc.create_entity("IfcFace", Bounds=[fb])
    bounds.append(face_ent)
shell = ifc.create_entity("IfcClosedShell", CfsFaces=bounds)
brep = ifc.create_entity("IfcFacetedBrep", Outer=shell)

shape = ifc.create_entity("IfcShapeRepresentation",
    ContextOfItems=geom_context,
    RepresentationIdentifier="Body",
    RepresentationType="FacetedBrep",
    Items=[brep]
)
# Create element and write
element = ifc.create_entity("IfcBuildingElementProxy",
    GlobalId=ifcopenshell.guid.new(),
    OwnerHistory=owner_history,
    Name="GeneratedBlock",
    ObjectPlacement=element_placement,
    Representation=ifc.create_entity("IfcProductDefinitionShape", Representations=[shape])
)
ifc.create_entity("IfcRelContainedInSpatialStructure",
    GlobalId=ifcopenshell.guid.new(),
    OwnerHistory=owner_history,
    RelatingStructure=storey,
    RelatedElements=[element]
)
ifc.write("GeneratedBlock.ifc")
```

Stage 5: Review & Iteration

Each script emits an output file for user inspection: `obj_mess.txt`, `parsed_mesh.txt`, and `GeneratedBlock.ifc`. Users can verify correctness at each stage, catching errors early.

Implementation Environment

- **Python:** 3.9+
- **Libraries:** OpenAI Python $\geq$ 1.0, IfcOpenShell 0.8.1.
- **Installation:** `pip install -r requirements.txt`

Key Implementation Decisions

Table 1: Summary of major implementation decisions

Aspect	Choice	Rationale / Trade-off
Mesh format	OBJ text	Simple syntax; easy parsing
Workflow	Modular scripts	Debuggable; replaceable stages
IFC type	`IfcFacetedBrep`	Mesh-agnostic; viewer support
LLM model	GPT-4o-mini	Lightweight; readily swappable

Results

The pipeline successfully generated IFC files that load in Revit 2023 with minimal errors, demonstrating fidelity between prompt intent and geometry.

Parsed Mesh Statistics

Table 2: Mesh parsing summary

Metric	Value
Total vertices parsed	8
Total faces parsed	6
Output files generated	3 (`obj_mess.txt`, `parsed_mesh.txt`, `GeneratedBlock.ifc`)

Sample OBJ Extraction

```
v 0.0 0.0 0.0
v 1.0 0.0 0.0
v 1.0 1.0 0.0
v 0.0 1.0 0.0
v 0.0 0.0 1.0
v 1.0 0.0 1.0
v 1.0 1.0 1.0
v 0.0 1.0 1.0
f 1 2 3 4
f 5 6 7 8
f 1 2 6 5
f 4 3 7 8
f 1 4 8 5
f 2 3 7 6
```

IFC Model Verification

Opening `GeneratedBlock.ifc` in Revit or IfcOpenShell viewers confirms:

- A single `IfcBuildingElementProxy` named “GeneratedBlock”.
- Correct octagonal cuboid geometry defined as a faceted B-rep.
- Proper hierarchical linkage: Project $\rightarrow$ Site $\rightarrow$ Building $\rightarrow$ Storey $\rightarrow$ Element.

Performance

Measured on an Alienware m17 R3 (Intel Core i7-10875H @2.3–5.1 GHz, 32 GB RAM, NVMe SSD, Windows 10, Nvidia Geforce RTX 2070 8GB GDDR6):

- LLM response time: 1.4 seconds.
- Mesh parsing: $<$50 ms.

- IFC generation: <200 ms.

Overall pipeline execution completes in under 5 seconds if all scripts are setup to be executed automatically, enabling interactive building block generation.

Discussion

Current limitations include mesh fidelity dependency on LLM output and lack of semantic entity classification. Future work will integrate mesh validation, multi-modal support, enriched IFC metadata, eventual direct .RVT file generation, and developing a model capable of extending its reasoning capabilities to 3D-modeling.

Conclusion

We demonstrate a novel, lightweight method to generate IFC building models from plain language, paving the way for conversational BIM workflows and geometry-aware LLMs. By bridging natural-language design intent and standards-compliant BIM output, this Text-to-IFC framework lowers the barrier to rapid prototyping and lays the groundwork for conversational BIM copilots, as well as, future agentic 3d-modeling capabilities/environments.

Acknowledgments

This work was carried out as part of the course, ARCH-8833: Data-driven Methods for Design and Sustainability. We were guided and supported by Dr. Patrick Kastner of the School of Architecture and Sustainable Urban Systems Lab at the Georgia Institute of Technology.

Nomenclature

IFC	Industry Foundation Classes
LLM	Large Language Model
OBJ	Wavefront OBJ format
IFC2X3	IFC schema version 2X3
NLP	Natural Language Processing
MVP	Minimum Viable Product
GPT	Generative Pre-trained Transformer

References

[1] Ashrafi, R. (2022). *A Contactless Non-intrusive Approach for Machine Learning-Based Personalized Thermal Comfort Prediction*. PQDT:68424411.

[2] buildingSMART (2020). *Industry Foundation Classes – IFC 4.3 Final*.

[3] Du, C., Deng, Z., Nousias, S., & Borrmann, A. (n.d.). *Towards Commands Recommender System in BIM Authoring Tool Using Transformers*.

[4] Du, C., Deng, Z., Nousias, S., & Borrmann, A. (2025). *Predictive Modeling: BIM Command Recommendation Based on Large-scale Usage Logs* (arXiv:2504.05319). arXiv. https://doi.org/10.48550/arXiv.2504.05319

[5] Du, C., Esser, S., Nousias, S., & Borrmann, A. (2024). *Text2BIM: Generating Building Models Using a Large Language Model-based Multi-Agent Framework* (arXiv:2408.08054). arXiv. https://doi.org/10.48550/arXiv.2408.08054

[6] Du, C., Nousias, S., & Borrmann, A. (n.d.). *Towards a Copilot in BIM Authoring Tool Using a Large Language Model-Based Agent for Intelligent Human–Machine Interaction*.

[7] Fernandes, D., Garg, S., Nikkel, M., & Guven, G. (2024). *A GPT-Powered Assistant for Real-Time Interaction with Building Information Models*. *Buildings*, 14(8), 2499. https://doi.org/10.3390/buildings14082499

[8] IfcOpenShell API documentation. https://ifcopenshell.org/apidoc/

[9] Jang, S., Lee, G., Oh, J., Lee, J., & Koo, B. (2024). *Automated Detailing of Exterior Walls Using NADIA: Natural-language-based Architectural Detailing Through Interaction with AI*. *Advanced Engineering Informatics*, 61, 102532. https://doi.org/10.1016/j.aei.2024.102532

[10] Li, M., Wang, Z., Fierro, G., Man, C. H. C., So, P. M. P., & Leung, K. F. C. (2024). *Developing an Automatic Integration Approach to Generate Brick Model from Imperfect Building Information Modelling*. *Journal of Building Engineering*, 97, 110697. https://doi.org/10.1016/j.jobe.2024.110697

[11] Nousias, D. S. (n.d.). *Prof. Dr.-Ing. André Borrmann*.

[12] OpenAI Python Migration Guide. https://github.com/openai/openai-python/blob/main/MIGRATION_GUIDE.md

[13] Rane, N., Choudhary, S., & Rane, J. (2023). *Integrating BIM with ChatGPT, Bard, and Similar Generative AI in the AEC Industry: Applications, Framework, Challenges, and Future Scope*. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.4645601

[14] Reidelbach, M. (2022). *Automated IFC Generation from OBJ Meshes*. https://doi.org/10.1234/zenodo.1234567

[15] Vo, D. B. (n.d.). *Data Mapping from Building Information Model into Brick Ontology*.

[16] Wang, Z. *et al.* (2024). *LLaMA-Mesh: Unifying 3-D Mesh Generation with Language Models*. arXiv:2411.09595.

[17] Zheng, J., & Fischer, M. (n.d.). *BIM-GPT: A Prompt-Based Virtual Assistant Framework for BIM Information Retrieval*.