Relatório Técnico Detalhado: Assistente e Validador BIM Unificado

Versão 1.0

Data: 10 de julho de 2025

Sumário

Este documento oferece uma análise técnica aprofundada da aplicação "Assistente e Validador BIM Unificado". O objetivo é descrever em detalhes a arquitetura do sistema, as tecnologias empregadas, o fluxo de execução de ponta a ponta e a lógica por trás de cada componente principal, tanto no backend quanto no frontend. A análise é baseada inteiramente no código-fonte e nos documentos contidos no repositório do projeto.

1. Introdução e Visão Geral

1.1. Contexto e Problema

A indústria de Arquitetura, Engenharia e Construção (AEC) enfrenta desafios significativos relacionados à interoperabilidade e à qualidade dos dados em modelos de Informação da Construção (BIM). O formato IFC (Industry Foundation Classes), embora seja um padrão aberto, frequentemente apresenta inconsistências. Conforme apontado na apresentação do projeto, **falhas na definição de relações espaciais e hierarquias** são uma fonte primária de erros, e **mais de 30% dos atrasos e custos extras** em projetos de construção estão ligados a problemas de modelagem e interpretação incorreta de dados. A validação manual é demorada, propensa a erros e não escalável.

1.2. Proposta de Solução

O "Assistente e Validador BIM Unificado" emerge como uma solução holística para este problema. Trata-se de uma aplicação web que centraliza e automatiza o ciclo de vida da validação e exploração de modelos BIM. Seus pilares são:

1. **Validação Automatizada:** Utiliza ontologias e regras formais (SHACL) para verificar a conformidade do modelo contra um padrão de qualidade predefinido.
2. **Diagnóstico Inteligente:** Emprega Modelos de Linguagem Avançados (LLMs) para analisar os conflitos detectados e fornecer sugestões de correção claras e contextuais.
3. **Exploração Interativa:** Permite que os usuários não apenas vejam os erros, mas também explorem o modelo de múltiplas formas: através de um **visualizador 3D**, de um **grafo de dados** e de um **chatbot** que entende linguagem natural.

O objetivo final é transformar o processo de validação de reativo e manual para proativo, inteligente e integrado.

1.3. Estrutura do Documento

Este relatório está dividido nas seguintes seções:

- **Arquitetura da Solução:** Descreve a estrutura geral do sistema, seus componentes e a organização do código.
- **Análise Detalhada do Backend:** Explora cada módulo de serviço, rotas da API e a lógica de negócio.
- **Análise Detalhada do Frontend:** Disseca a lógica da interface do usuário, visualização 3D e comunicação com o backend.
- **Fluxo de Dados de Ponta a Ponta:** Narra um caso de uso completo, desde o upload do arquivo até a consulta de dados.
- **Conclusão e Trabalhos Futuros:** Sumariza as conquistas do projeto e aponta direções para evolução.

2. Arquitetura da Solução

A aplicação é projetada com uma arquitetura de três camadas principais: Frontend (Cliente), Backend (Servidor) e Serviços Externos, orquestrados para funcionar de maneira coesa.

2.1. Visão Geral da Arquitetura

- **Frontend (Cliente):** Uma Single-Page Application (SPA) responsável por toda a interação com o usuário. Construída com HTML5, CSS3 e JavaScript moderno, ela renderiza a interface, o visualizador 3D e o grafo, e se comunica com o backend via requisições HTTP (AJAX/Fetch).
- **Backend (Servidor):** Um servidor web construído com **Flask (Python)**. Ele atua como o cérebro da operação, expondo uma API RESTful que o frontend consome. É responsável por:
    - Receber os arquivos IFC.
    - Processar a geometria 3D.
    - Executar o motor de validação.
    - Converter os dados para RDF.
    - Interagir com os serviços externos.
- **Serviços Externos:**
    - **Apache Jena Fuseki:** Um servidor de triplestore que armazena o modelo BIM como um grafo de conhecimento RDF e expõe um endpoint SPARQL para consultas semânticas.
    - **Ollama (Gemma3):** Um serviço que hospeda o LLM e fornece uma API para

gerar as sugestões de correção.

O docker-compose.yml garante que os serviços externos (Fuseki e Ollama) sejam executados em containers Docker, simplificando a configuração do ambiente e garantindo consistência.

2.2. Estrutura de Diretórios do Repositório

A organização do código-fonte reflete essa arquitetura modular :

```
ontology-BIM/
├── app/                      # Código-fonte da aplicação Flask
│   ├── __init__.py           # Inicializa a aplicação Flask
│   ├── routes.py             # Define as rotas (endpoints) da API
│   ├── services/             # Módulos com a lógica de negócio principal
│   │   ├── chatbot_logic.py      # Lógica do chatbot (NLU e SPARQL)
│   │   ├── fuseki_manager.py     # Gerencia a comunicação com o Fuseki
│   │   ├── validation_engine.py  # Motor de validação (SHACL e LLM)
│   │   └── ...
│   ├── static/               # Arquivos estáticos (CSS, JS)
│   │   ├── js/               # Módulos JavaScript (main.js, graph.js, etc.)
│   │   └── css/style.css
│   └── templates/            # Templates HTML (index.html)
├── data/                     # Arquivos de dados e ontologias
│   └── ifc-ontology.ttl      # Regras de validação em SHACL
├── nlu_model/                # Modelo de NLU treinado com spaCy
├── uploads/                  # Pasta temporária para arquivos .ifc
├── config.py                 # Configurações da aplicação (chaves, endpoints)
├── docker-compose.yml        # Orquestração dos serviços externos
├── requirements.txt          # Dependências Python
└── run.py                    # Script para iniciar o servidor Flask
```

3. Análise Detalhada do Backend (Servidor Flask)

O backend é o núcleo funcional do sistema. Ele processa os dados brutos do IFC e os transforma em insights e visualizações.

3.1. Ponto de Entrada e Rotas (app/routes.py)

Este arquivo define a API que o frontend consome. As rotas mais importantes são:

- */*: Renderiza a página principal index.html.

- **/validate (POST)**: Este é o endpoint central do fluxo de trabalho. Ao receber um arquivo .ifc:
    1. Salva o arquivo temporariamente.
    2. Chama extract_ifc_geometry para obter os dados 3D.
    3. Chama validation_engine.validate_model para obter o relatório de validação.
    4. Chama fuseki_manager.upload_to_fuseki para carregar o grafo de conhecimento.
    5. Retorna um JSON consolidado para o frontend com os resultados da validação, os dados 3D e os nós em conflito.
- **/ask (POST)**: Recebe uma pergunta do chatbot, passa para o chatbot_logic.py processar e retorna a resposta.
- **/graph-data (GET)**: Retorna um subgrafo de dados focado em um objeto específico, para visualização.
- **/api/full-graph (GET)**: Retorna o grafo completo (com limite) para a visualização inicial.
- **/ontology-summary (GET)**: Retorna uma lista de todos os tipos de objetos e relações presentes no grafo, usada para popular o "Construtor de Consultas".

3.2. Motor de Validação (services/validation_engine.py)

Este módulo é responsável por verificar a conformidade do modelo.

- **_populate_rdf_graph_for_validation**: Esta função cria um grafo RDF simplificado, contendo apenas as informações estritamente necessárias para as regras SHACL. Isso otimiza o processo de validação, que não precisa operar sobre o grafo completo.
- **validate_model**:
    1. Carrega o grafo de dados da função anterior.
    2. Carrega o grafo de regras do arquivo data/ifc-ontology.ttl.
    3. Executa pyshacl.validate, que compara os dados com as regras.
    4. Se houver violações, ele percorre o results_graph do pyshacl para extrair a mensagem de erro (sh:resultMessage) e o elemento que falhou (sh:focusNode).
    5. Para cada violação, chama _get_llm_suggestion.
- **_get_llm_suggestion**:
    1. Constrói um prompt específico, instruindo o LLM a agir como um especialista BIM e fornecer uma sugestão concisa. Ex: "Você é um especialista em BIM. Forneça uma sugestão... para o seguinte conflito: Paredes devem estar contidas em exatamente uma estrutura espacial".
    2. Envia este prompt para a API do Ollama (configurada em config.py).
    3. Processa a resposta JSON do LLM e a retorna para ser incluída no relatório

final.

3.3. Gerenciador do Grafo de Conhecimento (services/fuseki_manager.py)

Este módulo lida com a conversão IFC-para-RDF e a comunicação com o Apache Fuseki.

- **convert_ifc_to_rdf**: Esta é uma função complexa e crucial. Ela usa o ifcopenshell para iterar sobre **todas** as entidades e relacionamentos do arquivo IFC. Diferente do grafo de validação, este é completo e captura a riqueza semântica do modelo:
  - **IfcObjectDefinition**: Cria um nó para cada objeto e o associa à sua classe (ex: ex:123 rdf:type ifc:IfcWall).
  - **IfcRelAggregates**: Cria relações de agregação (ex: ifc:Building inst:aggregates ifc:Floor).
  - **IfcRelContainedInSpatialStructure**: Cria relações de contenção espacial (ex: ifc:Floor inst:contains ifc:Wall).
  - **IfcRelAssociatesMaterial**: Associa objetos aos seus materiais.
  - **IfcRelDefinesByProperties**: Extrai e associa propriedades e quantidades (como área, volume, etc.) aos objetos.
- **upload_to_fuseki**: Primeiro, envia uma requisição DELETE para limpar o grafo antigo no Fuseki. Em seguida, serializa o novo grafo gerado para o formato Turtle e o envia via POST para o endpoint GSP (Graph Store Protocol) do Fuseki.
- **get_ontology_summary**: Executa duas consultas SPARQL no Fuseki: uma para obter todos os tipos de classes (?s a ?type) e exemplos de seus nomes, e outra para obter todos os tipos de predicados (relações) usados no grafo.

3.4. Lógica do Chatbot (services/chatbot_logic.py)

Este módulo implementa a interface de linguagem natural.

- **_load_nlp_model**: Carrega o modelo spaCy treinado da pasta nlu_model. O treinamento, definido em setup_nlu.py, ensinou o modelo a reconhecer intenções como saudacao, despedida e perguntar_propriedade a partir de frases de exemplo.
- **extract_bim_object e extract_bim_property**: Usam expressões regulares (regex) para extrair as entidades da pergunta do usuário. Por exemplo, de "Qual a relação 'tipo' para o objeto 'parede-01'?", ele extrai tipo e parede-01.
- **query_specific_property**:
  1. Recebe o objeto e a propriedade extraídos.
  2. Mapeia a propriedade em linguagem natural para seu equivalente técnico no grafo (ex: "material" -> inst:hasMaterial).
  3. Constrói uma consulta SPARQL para buscar essa informação no Fuseki.
  4. Formata o resultado em uma frase de resposta para o usuário.
- **process_user_question**: Orquestra o fluxo. Recebe o texto do usuário, usa o

spaCy para obter a intenção e, se for perguntar_propriedade, chama as funções de extração e consulta para obter a resposta.

4. Análise Detalhada do Frontend (Interface Web)

O frontend é a porta de entrada para todas as funcionalidades do sistema, projetado para ser intuitivo e informativo.

4.1. Estrutura e Orquestração (index.html e js/main.js)

- **index.html**: Define a estrutura da página com divs para cada seção principal: upload, relatório, visualizador 3D, chatbot e grafo. Ele importa as bibliotecas Three.js e Vis.js via CDN e o script principal main.js como um módulo.
- **main.js**: É o script orquestrador.
  - No evento DOMContentLoaded, ele obtém referências para todos os elementos da UI e adiciona os event listeners.
  - O event listener do formulário de upload (uploadForm) é o gatilho principal. Ele previne o envio padrão, cria um FormData, chama a API de validação do backend (/validate), e, no retorno, chama as funções apropriadas para renderizar o relatório, o modelo 3D e o grafo.
  - Ele gerencia o estado da UI, como desabilitar botões durante o carregamento e exibir mensagens de status.

4.2. Comunicação com a API (js/api.js)

Este arquivo abstrai todas as chamadas fetch para o backend. Ele exporta funções como validateModel, askChatbot, getGraphData, etc. Isso torna o código em main.js mais limpo e focado na lógica da UI, enquanto a lógica de comunicação de rede fica isolada aqui.

4.3. Visualização 3D (Lógica em js/main.js)

- **setupThreeJs**: Inicializa os componentes essenciais do Three.js: a Scene (o mundo 3D), a PerspectiveCamera (o olho do observador), o WebGLRenderer (que desenha na tela), e o OrbitControls (que permite navegar com o mouse).
- **loadProcessedIFCModel**:
  1. Recebe o array elements_3d_data do backend.
  2. Itera sobre cada elemento. Para cada um, cria um THREE.BufferGeometry.
  3. Define os atributos position (com os vértices) e index (com os índices das faces) da geometria.
  4. Cria um THREE.Mesh (a combinação de geometria e material) e o adiciona à cena.
  5. **Crucialmente**, armazena cada mesh em um Map chamado guidToMeshMap, usando o GlobalId do elemento como chave. Isso permite acesso instantâneo

ao objeto 3D a partir de seu ID.

- **highlightValidationErrors**: Recebe os resultados da validação, itera sobre os conflitos, extrai o GlobalId de cada elemento com erro, usa o guidToMeshMap para encontrar o mesh correspondente e troca seu material pelo errorMaterial (vermelho).
- **onMouseClick**: Usa uma técnica chamada **raycasting**. Ele projeta um raio da posição da câmera através do ponto onde o mouse clicou na tela. Ele então verifica quais objetos 3D na cena são interceptados por esse raio. O primeiro objeto interceptado é considerado o selecionado, tem seu material trocado para destaque (amarelo) e suas informações são exibidas no chatbot. Além disso, ele chama highlightNodeInGraph para conectar a seleção 3D à visualização do grafo.

4.4. Visualização do Grafo (js/graph.js)

- **drawGraph**:
  1. Recebe os dados de nós e arestas do backend.
  2. Chama applyConflictHighlighting para pré-processar os dados.
  3. Inicializa um novo vis.Network no div do contêiner do grafo, passando os dados e um objeto de opções que define a aparência e o comportamento físico do grafo (como repulsão entre nós).
  4. Adiciona um event listener de doubleClick que permite ao usuário expandir a visualização ao clicar duas vezes em um nó, buscando seus vizinhos.
- **applyConflictHighlighting**:
  1. Recebe os dados do grafo e o objeto conflictMessages (que mapeia GlobalId para mensagem de erro).
  2. Itera sobre os nós do grafo. Se o ID de um nó corresponde a um ID em conflictMessages, ele altera a cor do nó para vermelho e define sua propriedade title para ser um HTML formatado com a mensagem de erro, que aparece como um tooltip.
- **highlightNodeInGraph**: Chamado a partir da interação 3D, ele usa a API da vis.Network para encontrar um nó pelo seu ID e aplicar a seleção (network.setSelection) e o foco (network.focus) nele.

5. Fluxo de Dados de Ponta a Ponta (Caso de Uso)

Vamos seguir o fluxo completo para um caso de uso: **O usuário carrega um modelo, identifica um erro em uma parede e pergunta qual o seu material.**

1. **Upload:** O usuário seleciona casa.ifc e clica em "Validar Modelo".
2. **Requisição:** main.js envia o arquivo para a rota /validate no backend.
3. **Processamento Backend:**
   - routes.py recebe o arquivo.

    - validation_engine.py é chamado. Ele cria um grafo RDF simplificado, o compara com ifc-ontology.ttl e descobre que uma parede não está contida em uma estrutura espacial (IfcSpatialStructureElement).
    - O motor de validação envia a mensagem de erro "Paredes devem estar contidas em..." para o Ollama e recebe de volta uma sugestão como "Verifique se a parede está associada a um andar (IfcBuildingStorey)".
    - extract_ifc_geometry processa a geometria de todos os elementos, incluindo a parede com erro.
    - fuseki_manager.py converte o IFC inteiro para um grafo RDF completo e o carrega no Fuseki.
4. **Resposta:** O backend retorna um JSON para o frontend contendo:
    - validation: O relatório com o conflito da parede e a sugestão da IA.
    - elements_3d_data: Os vértices e índices de todos os objetos.
    - conflicting_nodes: Uma lista com o GlobalId da parede com erro.
5. **Renderização Frontend:**
    - main.js recebe a resposta.
    - O relatório de validação é exibido na tela.
    - loadProcessedIFCModel desenha o modelo 3D.
    - highlightValidationErrors usa o GlobalId da parede para encontrá-la no guidToMeshMap e pintá-la de vermelho.
    - getFullGraph é chamado, e graph.js desenha o grafo de dados, também destacando o nó da parede em vermelho.
6. **Interação do Usuário:**
    - O usuário vê a parede vermelha, clica nela no modelo 3D. O chatbot exibe seu nome: "parede-interna-01".
    - O usuário digita no chat: "qual o material da 'parede-interna-01'?"
7. **Consulta ao Chatbot:**
    - A pergunta é enviada para a rota /ask.
    - chatbot_logic.py usa spaCy e identifica a intenção perguntar_propriedade.
    - Ele extrai "material" e "parede-interna-01".
    - Constrói uma consulta SPARQL: SELECT ?materialLabel WHERE { ?parede rdfs:label "parede-interna-01" . ?parede inst:hasMaterial ?material . ?material rdfs:label ?materialLabel . }.
    - A consulta é enviada ao Fuseki, que retorna "Tijolo Cerâmico".
8. **Resultado Final:** A resposta é enviada de volta ao frontend, que exibe no chat: "A propriedade 'material' para 'parede-interna-01' é: 'Tijolo Cerâmico'." O grafo também é atualizado para focar no nó da parede e seu material.

6. Conclusão e Trabalhos Futuros

6.1. Conclusão

O projeto **Assistente e Validador BIM Unificado** representa um avanço significativo na forma como os modelos BIM são validados e explorados. Ao integrar de forma inovadora tecnologias de Processamento de Dados BIM (ifcopenshell), Web Semântica (RDF, SHACL, SPARQL), Inteligência Artificial (spaCy, LLMs) e Visualização de Dados (Three.js, Vis.js), a aplicação consegue entregar uma solução robusta, intuitiva e poderosa.

A arquitetura modular, a separação clara de responsabilidades entre backend e frontend, e o uso de contêineres para serviços externos demonstram uma engenharia de software sólida e preparada para escalabilidade. O projeto não apenas resolve o problema prático de detecção de erros, mas também enriquece o processo ao fornecer sugestões inteligentes e múltiplas perspectivas de análise (visual, semântica e textual).

6.2. Trabalhos Futuros

A plataforma atual serve como uma base excelente para futuras expansões:

- **Correção Automatizada ou Semi-Automatizada:** Implementar funcionalidades que permitam ao usuário aplicar as sugestões da IA com um clique, modificando o grafo de conhecimento e, potencialmente, gerando um novo arquivo IFC corrigido.
- **Análises Mais Complexas:** Expandir o validation_engine com regras SHACL mais sofisticadas e adicionar novos módulos de serviço para outras análises de engenharia (ex: análise estrutural, verificação de normas de acessibilidade, etc.), similar ao já existente thermal_analysis.py.
- **Colaboração em Tempo Real:** Integrar tecnologias como WebSockets para permitir que múltiplos usuários visualizem e comentem sobre o mesmo modelo simultaneamente.
- **Histórico de Versões:** Salvar "snapshots" do grafo no Fuseki para permitir a comparação entre diferentes versões de um mesmo modelo IFC, destacando as alterações.
- **Personalização de Regras:** Criar uma interface onde o próprio usuário possa criar ou editar regras de validação SHACL sem precisar escrever o código diretamente.

Link do Relatório:
https://docs.google.com/document/d/1LGyeixvrZd9rgAs2AuseKhceVbco1ywRugJfljIdPso/edit?usp=sharing