AutoFEA: Enhancing AI Copilot by Integrating Finite Element Analysis Using Large Language Models with Graph Neural Networks

Shifu Hou[1,2], **Rick Johnson**[2], **Ramandeep Makhija**[2], **Lingwei Chen**[3], **Yanfang Ye**[1,2*]

[1]University of Notre Dame, Notre Dame, IN 46556
[2]Lucy Family Institute for Data & Society, Notre Dame, IN 46556
[3]Wright State University, Dayton, OH 45435
shou@nd.edu, rick.johnson@nd.edu, rmakhija@nd.edu, lingwei.chen@wright.edu, yye7@nd.edu

Abstract

Large Language Models (LLMs) have demonstrated significant potential across various applications, but their use as AI copilots in complex and specialized tasks is often hindered by AI hallucinations, where models generate outputs that seem plausible but are incorrect. To address this challenge, we develop AutoFEA, an intelligent system that integrates LLMs with Finite Element Analysis (FEA) to automate the generation of FEA input files. Our approach features a novel planning method and a graph convolutional network (GCN)-Transformer Link Prediction retrieval model, which enhances the accuracy and reliability of the generated simulations. The AutoFEA system proceeds with key steps: dataset preparation, step-by-step planning, GCN-Transformer Link Prediction retrieval, LLM-driven code generation, and simulation using CalculiX. In this workflow, the GCN-Transformer model predicts and retrieves relevant example codes based on relationships between different steps in the FEA process, guiding the LLM in generating accurate simulation codes. We validate AutoFEA using a specialized dataset of 512 meticulously prepared FEA projects, which provides a robust foundation for training and evaluation. Our results demonstrate that AutoFEA significantly reduces AI hallucinations by grounding LLM outputs in physically accurate simulation data, thereby improving the success rate and accuracy of FEA simulations and paving the way for future advancements in AI-assisted engineering tasks.

Introduction

In recent years, the development of LLMs has rapidly advanced, driving numerous innovations in the field of natural language processing (NLP). State-of-the-art models such as GPT-4o and Llama 3 have demonstrated exceptional capabilities in understanding and generating natural language texts (OpenAI 2023; Touvron et al. 2023). These models, leveraging deep neural network architectures and extensive training data, perform remarkably well across various tasks, including text generation, translation, summarization, and dialogue systems. As LLMs continue to evolve, their potential applications as AI copilots in industrial production are becoming increasingly apparent. By automating processes, optimizing production workflows, and providing intelligent decision support, LLMs can significantly enhance efficiency and precision in industrial settings. For instance, LLMs can aid in predicting maintenance needs for equipment, optimizing supply chain management, and offering real-time monitoring and recommendations in quality control (Trewin, Clarke, and Thomas 2020; Zhu, Lee, and Ko 2021). Additionally, LLMs can analyze production data to identify potential issues and optimization opportunities, thus improving the overall reliability and cost-effectiveness of production processes (Taylor and Smith 2021).

However, despite the significant advancements in LLMs, they still face the challenge of AI hallucination when dealing with complex and specialized tasks. AI hallucination refers to the generation of outputs that appear plausible but are incorrect or nonsensical (Bender et al. 2021). In industrial production, AI hallucinations can lead to erroneous decisions and operations, potentially causing production accidents or economic losses. For example, when LLMs are used to generate complex engineering designs or production plans, errors in the generated content can severely impact product quality and safety (Marcus and Davis 2020).

To address the issue of AI hallucinations, various approaches have been proposed to enhance the accuracy and reliability of language models. One such approach is knowledge augmentation, where external knowledge sources, such as knowledge graphs, are integrated with language models to provide a broader factual basis for reasoning (Petroni et al. 2019; Weissenborn, Kočiskỳ, and Dyer 2017). Another effective method is multi-modal integration, which combines text with other modalities, such as images, audio, or video, thereby providing additional context for generating more accurate outputs (Prakash, Chitta, and Geiger 2021; Li et al. 2019). Additionally, human-in-the-loop feedback loops represent another important strategy. In this approach, outputs generated by the model are reviewed and adjusted based on human feedback, which helps reduce errors and refine model performance (Christiano et al. 2017; Stiennon et al. 2020).

In addition to these methods, tool augmentation has emerged as a crucial approach for mitigating AI hallucinations. By allowing language models to interact with external tools such as code executors, calculators, or databases, models can generate outputs based on real computations or query results, thereby significantly reducing the incidence of hallucinations. The ReAct framework proposed by Yao et al. synergizes reasoning and action, enabling language models to

execute tasks by calling external APIs, thus improving both task completion and accuracy (Yao et al. 2022). Schick et al. explored how language models could autonomously learn to use tools, further enhancing their ability to handle complex tasks (Schick et al. 2024). Additionally, Gao et al. introduced the Program-Aided Language Models (PAL), which assist language models in complex reasoning tasks by generating code, resulting in more precise outputs (Gao et al. 2023).

Expanding on the concept of tool augmentation, we propose integrating FEA simulations into the LLM workflow as an effective way to mitigate AI hallucinations. By combining FEA, a precise physical simulation tool, with LLMs, we can not only validate the generated content with simulation results but also significantly reduce AI hallucinations in complex engineering tasks. This approach is particularly suited to scenarios requiring high precision and physical consistency, ensuring that the generated solutions are both linguistically coherent and physically accurate.

Although integrating FEA into LLM workflows has significant potential, this process faces substantial challenges, particularly in areas such as numerical solution convergence, multi-physics coupling, and more. To address these challenges, this paper proposes and develops an intelligent system named AutoFEA. First, we mitigate the complexity of traditional FEA software interfaces by using LLM-generated code. Traditional FEA software typically requires users to possess a high level of expertise and involves complex operations, posing a significant barrier to most non-expert users. By leveraging LLMs, we can automatically generate the corresponding simulation code based on the user's natural language description, simplifying the process and significantly lowering the entry barrier. Second, to handle complex FEA tasks, we introduce a step-by-step planning approach. Specifically, we decompose complex simulation problems into multiple simpler sub-problems, with each sub-problem corresponding to a step in the simulation process. This divide-and-conquer strategy not only reduces the complexity of the problem but also allows each step to be independently handled and optimized, thereby improving the overall accuracy and efficiency of the simulation. Finally, to further enhance the quality of the generated simulation code, we designed and implemented a link prediction model based on Graph Neural Networks (GNNs) and Transformers. This model analyzes and predicts the relationships between simulation steps, automatically retrieving the most relevant example codes from the training dataset, which are then used as references for generating new code. This approach allows us to effectively leverage existing simulation knowledge while ensuring that the generated code is accurate and operable. The key contributions of this paper are summarized as follows:

1. Creation of a comprehensive dataset of 512 FEA simulation projects, meticulously prepared for training and evaluating LLM-based FEA systems. To the best of our knowledge, this is the first dataset specifically designed for using LLMs to perform FEA simulations.
2. Development of the AutoFEA system, which effectively integrates FEA into LLM workflows, enabling the seamless execution of FEA simulations within an LLM framework. This integration not only simplifies the FEA process, particularly for non-experts, but also reduces the occurrence of AI hallucinations by grounding LLM outputs in physically accurate simulations.
3. Introduction of a novel planning approach that decomposes complex simulations into manageable steps, enhancing accuracy and efficiency by allowing each step to be independently optimized.
4. Implementation of a link prediction model that improves the success rate and accuracy of the generated simulation code by leveraging existing simulation examples, ensuring that the outputs are reliable and operable.

Proposed Method

In this section, we present the technical details of our proposed method AutoFEA for automating finite element analysis. Its overall framework is illustrated in Figure 1.

Step-by-Step Planning

In complex tasks like FEA, directly generating a complete code using a LLM can be challenging. The reason lies in the inherent complexity of these tasks, which often involve multiple interdependent steps, each requiring highly specialized knowledge and detailed operations. Attempting to generate the entire simulation code at once could lead to errors or omissions, as the LLM might struggle to manage the intricate dependencies and detailed requirements of each step. Therefore, decomposing the complex task into smaller, more manageable steps—known as step-by-step planning—is crucial. This approach simplifies the problem, enabling the LLM to generate code incrementally while ensuring that each step is accurate and feasible.

Code Segmentation and Step Descriptions In FEA software such as CalculiX and Abaqus, input files (inp files) are typically segmented into distinct code blocks using specific keywords like `*NODE` and `*ELEMENT`. These blocks represent different aspects of the simulation, such as node definitions, element definitions, material properties, and boundary conditions. Generally, multiple code blocks together constitute a complete step, where each step represents a particular phase or function of the simulation task.

In the training dataset, these code blocks are grouped based on their degree of relevance to corresponding steps. For each step, detailed descriptions are generated, including the specific objectives, parameter details, and implementation methods for that step. These pre-processed step descriptions, along with the corresponding code blocks, help the LLM learn how to decompose complex tasks and generate the associated code accurately.

Example Retrieval During the testing phase, when a user provides a new simulation task description, the system first generates an embedding vector of this description using the LLM. Next, the system searches the training dataset for the most similar description by comparing the embedding vec-

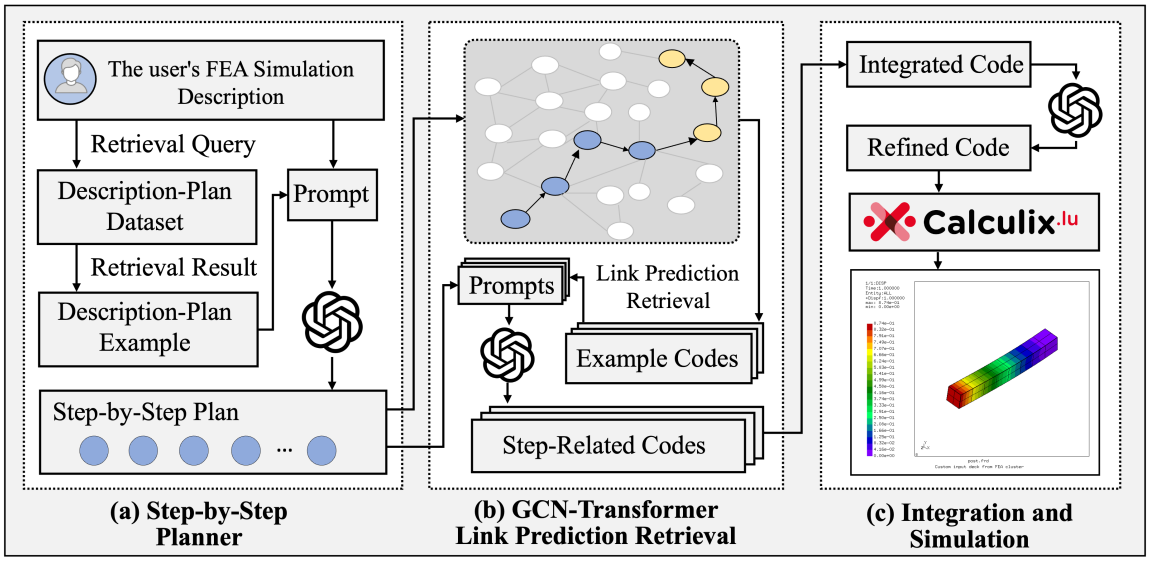

Figure 1: Overall framework for AutoFEA. (a) Step-by-Step Planning: Based on the user's description, a similar case from the training dataset is retrieved, along with its corresponding step-by-step plan, which is used as a few-shot prompt for the LLM to generate a new step-by-step plan. (b) GCN-Transformer Link Prediction Retrieval: For each step in the generated plan, the GNN performs link prediction to retrieve relevant nodes from a graph constructed from the training data. The retrieved node descriptions and codes serve as few-shot prompts to generate the corresponding code for each step. (c) Integration and Simulation: The generated codes are integrated and refined by the LLM, then passed to CalculiX for simulation. The simulation results provide feedback, helping to reduce AI hallucinations and ensuring reliable outputs for the AI copilot.

tors. This process is formalized as follows:

$$\text{sim}(\mathbf{Q}, \mathbf{D}_i) = \cos\left(\frac{E(\mathbf{Q}) \cdot E(\mathbf{D}_i)}{\|E(\mathbf{Q})\|\|E(\mathbf{D}_i)\|}\right) \quad (1)$$

Here, $\mathbf{Q}$ is the query embedding generated from the user's description, $\mathbf{D}_i$ represents the embedding of the i^{th} description in the training dataset, $E(\cdot)$ denotes the function that generates the embedding vector, and $\cos(\cdot)$ denotes the cosine similarity between the vectors. The description $\mathbf{D}_i$ with the highest similarity score is retrieved, along with its corresponding step-by-step plan.

This retrieved example, consisting of the description and its associated plan, serves as a few-shot example, which, combined with the simulation task description provided by the user, forms the input to the LLM, guiding it in generating a new plan tailored to the user's specific task. This retrieval-based approach ensures that the generated plan is grounded in similar successful cases, enhancing the accuracy, feasibility, and logical coherence of the resulting plan.

GCN-Transformer Link Prediction Retrieval

As discussed earlier, when generating code for each step in a FEA task, it's crucial to consider not just the current step's description, but also the code and descriptions of previous steps. This interdependence makes it challenging to retrieve relevant examples using a simple embedding-based similarity approach, which does not account for sequence dependencies. Therefore, we require a retrieval method that can effectively consider these sequential dependencies.

To address this, we initially considered using a Transformer-based model for retrieval. The Transformer architecture is well-suited for handling sequence data, as it captures complex dependencies between the current step and previous steps through its self-attention mechanism. By leveraging the Transformer, we can better incorporate contextual information and the sequential nature of the task, ensuring that the generated code logically aligns with the preceding steps. However, relying solely on the Transformer to capture sequence dependencies might overlook certain structural relationships inherent in the task. In our FEA code blocks, specific keywords such as `*DENSITY`, `*ELASTIC`, and `C3D20R` create natural links between similar steps, forming a graph structure. To capture these relationships, we introduced GNNs into the retrieval process. GNNs are capable of processing graph-structured data, allowing us to leverage the connections between nodes (steps) based on these keywords, thereby improving the retrieval of relevant historical steps and their associated code.

By integrating both Transformer and GNN approaches, we developed the GCN-Transformer Link Prediction model. This hybrid model not only processes the sequential information effectively but also utilizes the structural relationships within the graph. As a result, it can more accurately retrieve the most relevant steps and code blocks from the training dataset, ensuring that the generated code remains consistent with the task's overall context.

Graph Construction In our approach, the graph $G = (V, E)$ is constructed to represent the relationships between

different steps within the FEA process. Here, V denotes the set of nodes, where each node represents a step in the FEA workflow. The feature of each node is the embedding $\mathbf{h}_{v_i}^{(0)}$ of the step's description, generated by a LLM. The edges E represent the relationships between these steps, specifically indicating when two steps share a common keyword.

Formally, the graph is defined as follows:

$$V = \{v_1, v_2, \ldots, v_n\}, \quad \mathbf{h}_{v_i}^{(0)} \in \mathbb{R}^d \tag{2}$$

$$E = \{e_{ij} \mid v_i, v_j \in V \text{ and } \kappa(v_i, v_j) = 1\} \tag{3}$$

where $\kappa(v_i, v_j)$ is an indicator function that equals 1 if the steps corresponding to nodes v_i and v_j share at least one common keyword, and 0 otherwise. This graph structure effectively captures the relationships and dependencies between steps, forming the foundation for the GCN-Transformer Link Prediction model.

Link Prediction as Retrieval Task In FEA simulations, the input files for software like CalculiX and Abaqus often contain initial code blocks that describe the structure and mesh information of the simulation object. These parts of the code are typically generated through pre-processing tools, as accurately describing them in natural language is challenging. Therefore, these initial steps in the simulation are treated as known information.

Within our GCN-Transformer model, these known simulation steps can be viewed as an existing path $P = \{v_1, v_2, \ldots, v_k\}$ in the graph $G = (V, E)$, where each node v_i represents a known step in the simulation workflow. The task then is to find the most appropriate node v_{k+1} in the graph, given the current path P and the description $\mathbf{d}_{k+1}$ of the next step. Consequently, this retrieval task is framed as a link prediction problem. Formally, our goal is to identify the node $v_{k+1} \in V$ that maximizes the following function, forming a new edge $e_{k,k+1}$ with the existing path P and the step description $\mathbf{d}_{k+1}$:

$$v_{k+1} = \arg\max_{v \in V} \text{Score}(P, v, \Phi(\mathbf{d}_{k+1})) \tag{4}$$

where $\Phi(\mathbf{d}_{k+1})$ represents the embedding of the step description $\mathbf{d}_{k+1}$, generated by a LLM. The scoring function $\text{Score}(P, v, \Phi(\mathbf{d}_{k+1}))$ evaluates the compatibility between the current path P and the potential next step node v_{k+1}. This function combines information from the graph structure (captured by the GCN) and the sequence dependencies (captured by the Transformer), ensuring that the prediction accounts for the complex relationships between the steps in the simulation workflow.

By iteratively repeating this process, we can sequentially identify the most suitable nodes in the graph, ultimately constructing a complete path P^* that effectively guides the LLM in generating the simulation code. This method enables the model to consider known steps while dynamically adapting to the description of each new step, ensuring that the generated code is logically consistent with the overall requirements of the simulation task.

GCN-Transformer Model Our GCN-Transformer model operates on the Steps Graph, which includes all steps from the training dataset as well as the known structure and mesh-related steps from the testing dataset. This graph $G = (V, E)$ encompasses all possible steps v_i, each initialized with a feature embedding $\mathbf{h}_{v_i}^{(0)}$ generated by an LLM from the step description $\mathbf{d}_i$, such that $\mathbf{h}_{v_i}^{(0)} = \Phi(\mathbf{d}_i)$, where Φ represents the LLM embedding process.

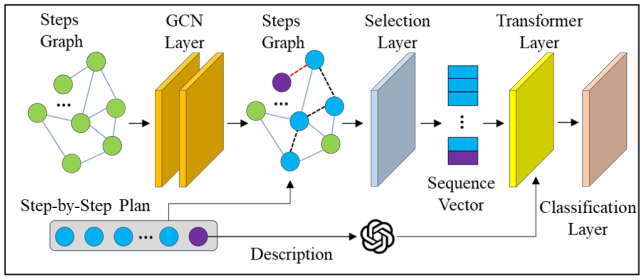

Figure 2: GCN-Transformer Model.

First, the model takes these initial node embeddings and inputs them into a Graph Convolutional Network (GCN) to effectively capture both local and global information within the graph structure. The forward propagation in the GCN is defined as follows:

$$\mathbf{h}_v^{(k)} = \sigma\left(\sum_{u \in \mathcal{N}(v) \cup \{v\}} \frac{1}{c_{uv}} \mathbf{W}^{(k)} \mathbf{h}_u^{(k-1)}\right) \tag{5}$$

where $\mathbf{h}_v^{(k)}$ represents the feature representation of node v at the k-th layer, $\mathcal{N}(v)$ denotes the set of neighboring nodes of v, c_{uv} is a normalization factor, $\mathbf{W}^{(k)}$ is the weight matrix for the k-th layer, and σ is an activation function. After several layers of GCN, the model outputs enriched node representations $\mathbf{h}_v^{(K)}$, which capture both the local and global contexts, enabling the model to uncover intricate relationships and patterns within the graph.

Next, these enriched node embeddings are combined into a sequence vector representing the path $P = \{v_1, v_2, \ldots, v_k\}$. The path matrix $\mathbf{P}_K$ is constructed by stacking the embeddings of the nodes along the path:

$$\mathbf{P}_K = \left(\begin{bmatrix} \mathbf{h}_{v_1}^{(K)T} & \mathbf{h}_{v_2}^{(K)T} & \cdots & \mathbf{h}_{v_k}^{(K)T} \end{bmatrix}\right)^T \tag{6}$$

This path vector $\mathbf{P}_K$ serves as the key K and value V in the attention mechanism, while the query comes from the embedding of the current step description $\Phi(\mathbf{d}_{k+1})$ generated by the LLM. In this respect, the attention mechanism can be computed as follows:

$$\text{Attention}(\Phi(\mathbf{d}_{k+1}), \mathbf{P}_K^T) = \text{softmax}\left(\frac{\Phi(\mathbf{d}_{k+1})\mathbf{P}_K^T}{\sqrt{d_{\mathbf{P}_K}}}\right)\mathbf{P}_K \tag{7}$$

where $d_{\mathbf{P}_K}$ represents the dimensionality of $\mathbf{P}_K$.

This process evaluates the similarity between the current step description and the path vector, using this similarity to weight the path vector and produce the final attention output. By iteratively applying this process, the model dynamically extends the path within the Steps Graph, ultimately

constructing a complete path P^* that effectively guides the LLM in generating the simulation code that is logically consistent with the overall simulation task.

Negative Sampling Strategy In link prediction tasks, the selection of negative samples is a critical consideration, particularly in complex graph structures. Randomly selecting negative samples can lead to the inclusion of nodes that are highly similar to the positive examples, which is especially detrimental in our scenario. Specifically, if a negative sample shares the same keywords as the target node v_{k+1}, such a node may have features that closely resemble those of the positive example, making it challenging for the model to distinguish between positive and negative cases, thereby compromising the learning process.

To address this issue, our approach employs a more stringent strategy for negative sampling. We only select nodes that share no common keywords with the target node v_{k+1} as negative samples. This means that we exclude any node from the graph G that is adjacent to v_{k+1} from being considered as a negative sample, since adjacent nodes typically share one or more keywords. By adopting this strategy, we effectively avoid the problem of false negatives and ensure that the model can more accurately learn to differentiate between positive and negative examples.

Integration and Simulation

Based on the optimal path P^* obtained from the previous steps, we proceed to generate the complete simulation code. For each node along the path P^* that is not part of the known nodes (i.e., nodes that are not related to structure and mesh information), we treat the node's description and the corresponding code as a few-shot example. This few-shot example, combined with the description and current code for the step being processed, is then fed into the LLM to generate the corresponding code for that step. This process is repeated for all steps along the path P^* until the complete simulation code is fully obtained.

After generating the full simulation code, we perform a final check by inputting the complete code along with the user's original query back into the LLM. This allows the LLM to review and potentially modify the code to ensure it aligns with the user's requirements and expectations. Finally, the validated simulation code is submitted to CalculiX for execution. The results of this simulation serve as a reference for the AI-copilot, thereby helping to reduce AI hallucinations by grounding the LLM's responses in actual simulation outcomes. This approach ensures that the AI-copilot provides accurate and reliable support in complex simulation tasks, enhancing the overall reliability of the system.

Experimental Results and Analysis

In this section, we provide a comprehensive analysis of our experimental results. We begin by detailing the process of constructing our specialized dataset, which serves as the foundation for training and evaluating our LLM-based FEA system. Following this, we conduct a thorough performance comparison, highlighting the effectiveness and advantages of our proposed method in comparison to other approaches. Finally, we present a detailed case study that demonstrates the seamless integration of FEA into the LLM workflow, showcasing the practical applicability and potential of our system in real-world scenarios.

Experimental Setup

Implementation The experiments were conducted on a machine equipped with a 12th Gen Intel(R) Core(TM) i9-12900K CPU, an NVIDIA RTX A6000 GPU, and 64 GB of RAM. The operating system used was Ubuntu 20.04, and the Python version was 3.10.12. For interacting with the LLM, we utilized the `langchain` library, version 0.2.29. The primary LLM used in our experiments was `GPT-4o`. Additionally, for generating embeddings, we employed `text-embedding-3-small`. Different LLM alternatives were evaluated to assess their impact on the performance of AutoFEA.

Data Collection and Preparation The dataset used in our experiments was collected from the official CalculiX website's test examples, which serve as a standard benchmark for verifying the functionality and performance of the CalculiX FEA software. They cover a wide range of simulation scenarios to test various types of analyses effectively.

Due to the high computational cost of running lengthy codes on `GPT-4o`, we excluded a portion of simulation projects with excessively long codes, resulting in 512 selected projects[1]. Each project includes a brief description at the beginning, such as "dynamic response to a linear force; no damping; DIRECT=NO" for the "beam15" project. These concise descriptions are insufficient to fully reproduce the entire simulation process, necessitating the generation of more detailed descriptions. Thus, we utilized `GPT-4o` to generate comprehensive descriptions for each project, which were then manually reviewed to ensure they accurately reflected the corresponding simulation scenarios.

Next, we segmented these projects into multiple steps based on four criteria outlined in Table 1. The segmented code for each step was then submitted to `GPT-4o`, which generated detailed descriptions for each step. This process resulted in 4,792 steps across the 512 projects, with an average of 9.36 steps per project. The number of steps per project ranged from a minimum of 3 to a maximum of 16.

Performance Comparison

The primary evaluation metric for our experiments is whether the generated code can successfully perform the simulation. To determine the success of a simulation, we compare the `dat` files generated by running the original `inp` code and the generated code. The comparison is based on two tolerance levels: a relative tolerance set to 1e-4 and an absolute tolerance set to 1e-6.

Relative tolerance defines the permissible difference between two values as a proportion of the larger value, meaning that this difference should not exceed 1e-4 of the larger value. Absolute tolerance, on the other hand, specifies a fixed threshold, ensuring that the difference between the two

[1]https://github.com/autufea/Autofea

Criteria	Description
Comment	The initial comments and *HEADING blocks are segmented as the first step, providing a simple explanation of the project.
Similar Tasks	Grouping consecutive code blocks that perform similar tasks based on identical starting keywords. For example, if multiple consecutive code blocks start with the keyword `*ELEMENT`, they are categorized into the same step.
Nested Blocks	Code blocks nested within higher-level code blocks are grouped together. A common example is: `*MATERIAL, NAME=ALUM`, followed by `*ELASTIC` and `*DENSITY`.
Enclosed by *STEP	Code segments enclosed by `*STEP` and `*END STEP` are treated as a single step.

Table 1: Segmentation Criteria for Project Steps

values remains within 1e-6, regardless of their magnitude. By applying these thresholds, we ensure that the generated simulation results are virtually indistinguishable from those produced by the original code.

We conducted five sets of experiments, where 90% of the data was randomly selected as the training set and 10% as the test set, repeated three times. The experiments are summarized as follows:

1. **Direct Generation**: Directly generating code from the description alone.
2. **Retrieval-Augmented Generation**: Generating code using the description and the retrieved corresponding code.
3. **Planning-Based Generation**: Generating a step-by-step plan from the description that guide the code generation.
4. **Retrieval-Augmented Planning**: Generating a step-by-step plan from the description, retrieving similar code based on the plan, and then generating the code.
5. **AutoFEA (Our Method)**: The proposed method that integrates the entire pipeline as described.

Additionally, it is important to note that although we removed excessively long codes from the dataset, directly using these codes without modification would still result in prohibitively long inputs. This poses two challenges: 1) the cost of processing such long inputs is high, and 2) the context window of the LLM may be insufficient to handle such long sequences. To mitigate these issues, for code blocks related to structure and mesh that exceed 10 lines, we omitted the excess content. The performance comparison of these methods is illustrated in Figure 3, which shows the success rates across the five different approaches. As depicted, our proposed method significantly outperforms the others, achieving the highest success rate in simulation accuracy.

Ablation Study

To assess the impact of LLM choice on AutoFEA's performance, we replaced `GPT-4o` with other LLM alternatives to rigorously test the framework. As reported in

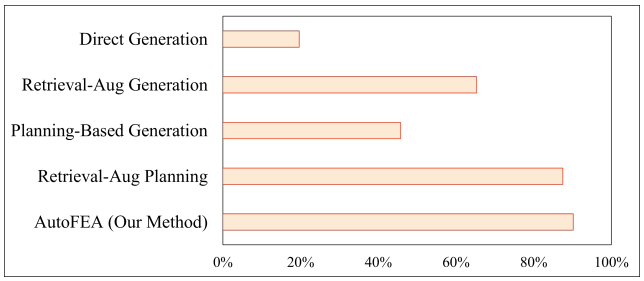

Figure 3: Performance comparison (success rate %).

Table 2, the results reveal performance variation among LLMs, but AutoFEA with varying LLMs consistently outperforms other conventional frameworks like direct generation, planning-based generation, and retrieval-augmented generation. These findings reaffirm AutoFEA's adaptability to different model strengths while maintaining robust performance. It is worth noting that the lack of GNN analysis is not an oversight but a deliberate choice. While more complex GNNs could enhance performance, we selected a two-layer GCN to emphasize the role of graph structure in retrieving relevant code examples, striking a balance between simplicity and functionality in meeting our learning goal.

Language Model	Success Rate (%)
GPT-4o	90.2
GPT-4o-mini	81.6
Gemini-Flash	75.2
Gemini-1.5-Pro	83.8
Llama 3 8b	78.8

Table 2: Comparison across different LLMs.

Case Study

To demonstrate the practical application of our method, we conducted a case study using FEA of a cantilever beam. The objective was to simulate the beam's response to shear force and analyze the displacement at the point farthest from the fixed end. As shown in Figure 4, the process begins with providing the FEA description to the AI system.

Initially, `GPT-4o` calculated the displacement at the free end of the beam to be approximately 0.0975 units based on the user's input. However, this result was derived under the assumption of linear deformation and failed to account for the material's plastic deformation. In this scenario, plastic deformation plays a more significant role in determining the displacement, leading to the AI hallucination observed in `GPT-4o`'s output. Such an oversight can have serious consequences, especially in engineering contexts where inaccurate predictions could lead to design flaws, structural failures, or even safety hazards. If these AI-generated results were used in real-world applications without verification, they could result in significant financial losses, legal liabilities, or pose significant risks to human safety.

To address this issue, we ran a complete FEA simulation through the AutoFEA system, incorporating the effects of plastic deformation, and fed the simulation results back into the system. When queried again, `GPT-4o` accurately identified the correct displacement value of approximately 0.8738 units, corresponding to Nodes 5 and 6. This case study highlights the importance of integrating FEA simulations into the AI reasoning process, as it helps avoid AI hallucinations caused by overlooking critical physical phenomena such as plastic deformation, thereby ensuring accuracy and reliability in complex engineering tasks.

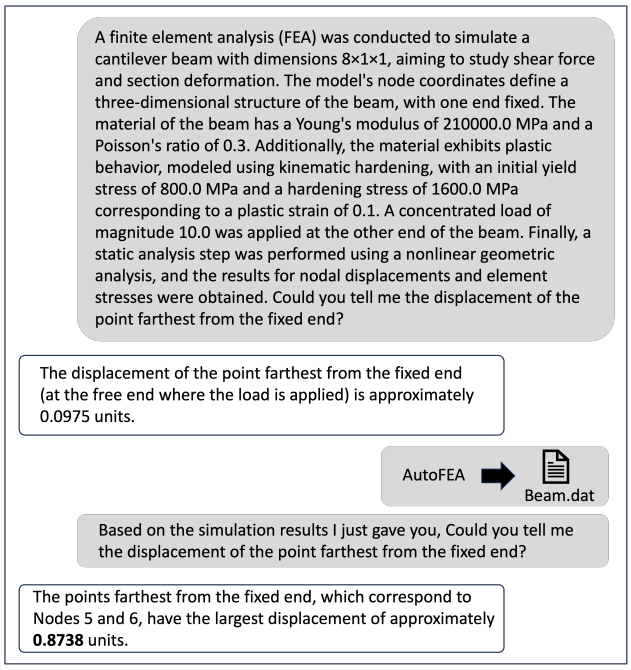

Figure 4: Case study: integrating FEA into AI reasoning.

Related Work

This section focuses on three key areas: strategies for mitigating AI hallucinations, automation of FEA model generation, and enhancement of retrieval tasks using GNNs.

Mitigating AI Hallucinations The growing application of LLMs across various domains (Brown et al. 2020; Radford et al. 2019; Yang et al. 2019) has highlighted the persistent challenge of AI hallucinations, where models produce plausible yet incorrect outputs. Several strategies have been proposed to address this issue. One common approach is knowledge augmentation, which integrates external knowledge bases or real-time data sources with the model to ensure more accurate content generation. Petroni et al. (2019) demonstrated that integrating knowledge bases with language models significantly reduces the likelihood of generating incorrect outputs. Peters et al. (2019) improved the accuracy of model outputs by enhancing contextual word representations through knowledge integration. Another significant strategy is multimodal integration, which combines text with other modalities such as images, audio, or video to create a more contextually enriched environment, thereby reducing the likelihood of generating incorrect information. Li et al. (2019) showed that integrating visual information with language models can reduce errors in generation tasks. Human-in-the-loop feedback loops represent another critical strategy for mitigating AI hallucinations. Christiano et al. (2017) proposed adjusting model outputs through human feedback to improve the accuracy of the generated content. Stiennon et al. (2020) demonstrated how learning from human feedback can optimize the summarization process.

Automation of FEA The automation of FEA model generation has garnered increasing attention for its potential to reduce manual labor and enhance accuracy in engineering simulations. Aizawa (1991) explored integrating CAD and CAE systems to streamline the FEA model generation process. Pantoja-Rosero, Achanta, and Beyer (2024) used computer vision and machine learning to automate FEA models from images, specifically for masonry buildings. Jia et al. (2022) developed a method combining BIM with ontology technology to automate the conversion of BIM data into FEA models, thus improving efficiency in structural design. Advancements in 3D FEA model automation were also discussed, with methods to reduce manual effort through specialized meshing techniques (Straughan et al. 2023).

Enhancing Retrieval with GNNs GNNs have been increasingly applied to enhance retrieval tasks, especially in complex data environments. Recent approaches such as GNN-RAG integrate the reasoning capabilities of GNNs with the language understanding of LLMs to improve the retrieval of relevant subgraphs in knowledge graph question answering (KGQA) tasks (Mavromatis and Karypis 2024). Similarly, GNN-Ret utilizes GNNs to enhance passage retrieval by considering semantic relationships between passages in question-answering systems (Li et al. 2024). Additionally, the use of GNNs in document retrieval has shown promise, particularly in generating concept maps from unstructured texts to capture semantic relationships and improve retrieval accuracy (Cui et al. 2022).

Conclusion

In this paper, we presented a novel approach for integrating LLMs with FEA to automate the generation of FEA input files. By leveraging a combination of step-by-step planning, GCN-Transformer link prediction retrieval models, and automated simulation, our method effectively addresses the challenges of complexity and accuracy inherent in FEA tasks, significantly reducing the occurrence of AI hallucinations. The proposed AutoFEA system not only simplifies the traditionally complex FEA process but also enhances the reliability and accuracy of the generated models. Our experiments demonstrated the effectiveness of this approach across various scenarios, providing a solid foundation for future developments in AI-assisted engineering workflows. We believe that our work opens new avenues for more seamless integration of LLMs into specialized domains, offering a reliable AI copilot for complex engineering tasks with greater accuracy and efficiency.

Acknowledgements

This work was partially supported by the NSF under grants IIS-2321504, IIS-2334193, IIS-2203262, IIS-2217239, CNS-2426514, CNS-2203261, and CMMI-2146076. S. Hou's work was partially supported by the Lucy Family Institute and AeTL's postdoc fund. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

Aizawa, T. 1991. Automatic Generation of Finite Element Modeling for Integrated CAD and CAE. In *CAD/CAM Robotics and Factories of the Future'90: Volume 2: Flexible Automation 5th International Conference on CAD/CAM, Robotics and Factories of the Future (CARS and FOF'90) Proceedings*, 273–278. Springer.

Bender, E. M.; Gebru, T.; McMillan-Major, A.; and Shmitchell, S. 2021. On the Dangers of Stochastic Parrots: Can Language Models Be Too Big? *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, 610–623.

Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language models are few-shot learners. *NeurIPS*, 33: 1877–1901.

Christiano, P. F.; Leike, J.; Brown, T.; Martic, M.; Legg, S.; and Amodei, D. 2017. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30.

Cui, H.; Lu, J.; Ge, Y.; and Yang, C. 2022. How can graph neural networks help document retrieval: A case study on cord19 with concept map generation. In *European Conference on Information Retrieval*, 75–83. Springer.

Gao, L.; Madaan, A.; Zhou, S.; Alon, U.; Liu, P.; Yang, Y.; Callan, J.; and Neubig, G. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, 10764–10799. PMLR.

Jia, J.; Gao, J.; Wang, W.; Ma, L.; Li, J.; and Zhang, Z. 2022. An automatic generation method of finite element model based on BIM and Ontology. *Buildings*, 12(11): 1949.

Li, L. H.; Yatskar, M.; Yin, D.; Hsieh, C.-J.; and Chang, K.-W. 2019. Visualbert: A simple and performant baseline for vision and language. *arXiv preprint arXiv:1908.03557*.

Li, Z.; Guo, Q.; Shao, J.; Song, L.; Bian, J.; Zhang, J.; and Wang, R. 2024. Graph Neural Network Enhanced Retrieval for Question Answering of LLMs. *arXiv preprint arXiv:2406.06572*.

Marcus, G.; and Davis, E. 2020. Rebooting AI: Building Artificial Intelligence We Can Trust. *Penguin Random House*, 55–70.

Mavromatis, C.; and Karypis, G. 2024. GNN-RAG: Graph Neural Retrieval for Large Language Model Reasoning. *arXiv preprint arXiv:2405.20139*.

OpenAI. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774*.

Pantoja-Rosero, B. G.; Achanta, R.; and Beyer, K. 2024. Automated image-based generation of finite element models for masonry buildings. *Bulletin of Earthquake Engineering*, 22(7): 3441–3469.

Peters, M. E.; Neumann, M.; Logan IV, R. L.; Schwartz, R.; Joshi, V.; Singh, S.; and Smith, N. A. 2019. Knowledge enhanced contextual word representations. *arXiv preprint arXiv:1909.04164*.

Petroni, F.; Rocktäschel, T.; Lewis, P.; Bakhtin, A.; Wu, Y.; Miller, A. H.; and Riedel, S. 2019. Language models as knowledge bases? *arXiv preprint arXiv:1909.01066*.

Prakash, A.; Chitta, K.; and Geiger, A. 2021. Multi-modal fusion transformer for end-to-end autonomous driving. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 7077–7087.

Radford, A.; Wu, J.; Child, R.; Luan, D.; Amodei, D.; and Sutskever, I. 2019. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8): 9.

Schick, T.; Dwivedi-Yu, J.; Dessì, R.; Raileanu, R.; Lomeli, M.; Hambro, E.; Zettlemoyer, L.; Cancedda, N.; and Scialom, T. 2024. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36.

Stiennon, N.; Ouyang, L.; Wu, J.; Ziegler, D.; Lowe, R.; Voss, C.; Radford, A.; Amodei, D.; and Christiano, P. F. 2020. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33: 3008–3021.

Straughan, R.; Kadry, K.; Parikh, S. A.; Edelman, E. R.; and Nezami, F. R. 2023. Fully automated construction of three-dimensional finite element simulations from Optical Coherence Tomography. *Computers in Biology and Medicine*, 165: 107341.

Taylor, Z.; and Smith, A. 2021. Applications of AI in Industrial Production. *IEEE Transactions on Industrial Informatics*, 17: 345–355.

Touvron, H.; Lavril, T.; Izacard, G.; Martinet, X.; Lachaux, M.-A.; Lacroix, T.; Rozière, B.; Goyal, N.; Hambro, E.; Azhar, F.; Rodriguez, A.; Joulin, A.; Grave, E.; and Lample, G. 2023. LLaMA: Open and Efficient Foundation Language Models. *arXiv preprint arXiv:2302.13971*.

Trewin, S.; Clarke, C.; and Thomas, J. 2020. AI in Scientific Computing. *Journal of Computational Science*, 45: 101–112.

Weissenborn, D.; Kočiskỳ, T.; and Dyer, C. 2017. Dynamic integration of background knowledge in neural nlu systems. *arXiv preprint arXiv:1706.02596*.

Yang, Z.; Dai, Z.; Yang, Y.; Carbonell, J.; Salakhutdinov, R. R.; and Le, Q. V. 2019. XLNet: Generalized autoregressive pretraining for language understanding. In *Advances in neural information processing systems*, 5754–5764.

Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; and Cao, Y. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Zhu, H.; Lee, Y.; and Ko, J. 2021. Automated Supply Chain Management with AI. *International Journal of Production Research*, 59: 105–120.