Equans MEP

Group 1 - Achilleas Leivadiotis, Beniamin Pasak, Nathan Bouquet, Vince Kotsis,
Jakub Suszwedyk, Sascha Zwietink, Anton Zegelaar

Department of Advanced Computing Sciences
Faculty of Science and Engineering
Maastricht University

Maastricht, The Netherlands

***Abstract*—We present a proof-of-concept pipeline for automating the placement of Mechanical, Electrical, and Plumbing (MEP) components in 3D building designs. Rather than relying on entirely manual efforts, our approach merges context-driven neural networks to reduce modeling time.**

Our pipeline begins by extracting and organizing data from Autodesk Revit exports, dividing each building model into "units" and linking each unit to a JSON file (component data) and an .obj mesh. The unit data, including adjacency relationships, is then normalized and embedded into vectors suited for neural network processing.

Our method consists of two neural network types: A type predictor that selects which component to place, followed by a location predictor that provides with the exact xyz coordinates. These neural networks are enhanced by Convolutional neural networks (CNNs), and further guided by a "context window" of surrounding units. Alongside, we have incorporated an expert system.

Experimental evaluations across multiple building datasets indicate that context-informed CNNs achieve consistently lower placement errors. Meanwhile, the expert system, though less precise, remains transparent and provides a baseline for interpretable predictions. Altogether, this proof-of-concept demonstrates how automated MEP placement can help address an industry-wide need for speed, accuracy, and cost-effectiveness in modern construction workflows.

CONTENTS

I. INTRODUCTION

The global construction industry is increasingly adopting modular building techniques, where structures are assembled from small prefabricated units built in large factories. This

trend is driven by the need for faster construction times and cost-effective buildings, especially in high-demand markets like the Netherlands, where there is a housing shortage. (Dita Hořínková, 2021) These modular buildings, designed for 10-25 years of use, are quick to construct but present a lot of pre-construction work, particularly in terms of integrating Mechanical, Electrical, and Plumbing (MEP) components to the building, which are essential for making the buildings operational. (Kazeem KO, 2024)

For modern buildings manual modeling and MEP coordination can consume over 50% of the project resources. (Teo, 2022) That's because the traditional process of adding MEP components to a building model is manual, labor-intensive, and time-consuming, (Kazeem KO, 2024) which adds to project costs and delays. This problem is especially significant when dealing with modular buildings, where MEP systems must be customized and coordinated across many small, interconnected units, although the units are very similar.

Given the rising demand for more efficient construction methods, automation of these processes becomes essential. Current Building Information Modeling (BIM) systems like Autodesk Revit allow for efficient modeling, but they still rely heavily on manual inputs for MEP components. The challenge lies in developing tools that can autonomously generate MEP models, reducing human error, and improving time efficiency while maintaining compliance with engineering standards and norms.

In this project, in collaboration with Equans, our aim is to develop an AI system capable of autonomously generating a 3D MEP model from an architectural BIM model. By automating the MEP design process, the system will not only enhance the speed of the design process but will also help address the shortage of skilled 3D modelers. This could potentially reduce MEP design times, leading to significant cost savings and faster project completion times.

We are doing that by answering two research questions.

- **How does the performance of the rule-based expert system compare with the AI-driven approach for component location prediction?**
- **What are the challenges and limitations in implementing AI-driven generation systems for MEP components?**

II. Approach

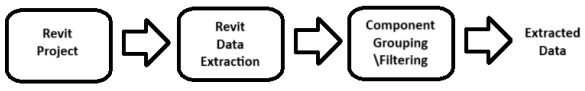

Fig. 1. Revit Data Extraction Pipeline

The first step in our approach to automating the placement of MEP components is extracting project data from Revit, a widely used BIM software. This process begins with a manual export, where the currently active view in Revit is saved as an .obj file. This is later used in a custom program designed to process the exported data and extract all necessary information for both training and applying our AI solution. This method ensures that we capture every placed component, along with the detailed structure of the building.

Following this stage, we filter out the components in order to obtain the components of desired properties (e.g. electrical components) and proceed to group the components that always appear together into new composite components, facilitating easier data processing and analysis.

Once extracted, the data is organized by dividing the project into individual units. Each unit is then represented by two files, which serve as inputs for our program. The first is a JSON file that contains detailed information about the unit: the placement and relative positions of its components, connections to adjacent units via their IDs, as well as the position of the unit relative to the whole building.

The second is a new .obj file that provides the unit's mesh with all components removed, offering a clear representation of the architectural geometry.

Fig. 2. Neural Network Model Pipeline

The exported files undergo further processing to prepare them for use in our system. The .obj file is converted into a three-dimensional tensor of points, which is then sampled randomly with probabilities weighted by the size of the faces from which the points are sampled. To achieve this we leverage the capabilities of the PyTorch3D library ((Ravi et al., 2020)).

Simultaneously, the JSON file is parsed and transformed into an instance of a Python unit class. This custom class organizes and structures all the unit's data in a clear and easily-accessible manner and implements methods allowing for easy data normalization.

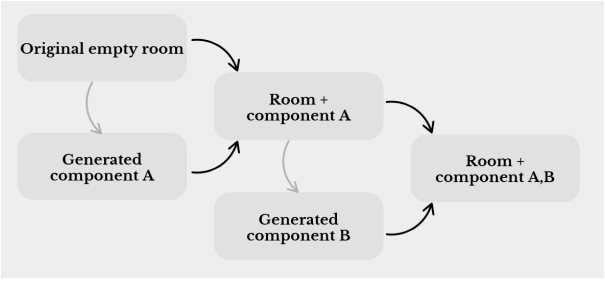

Fig. 3. Component-by-Component MEP component generation within a unit

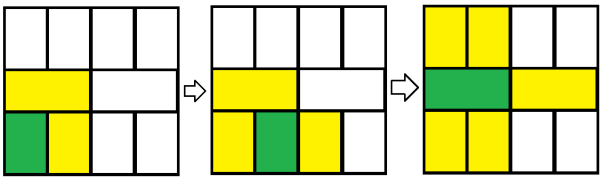

Fig. 4. unit-by-unit MEP component generation with a context window

Using the neural networks developed in PyTorch ((Paszke et al., 2019)) to generate components, first the data is retrieved, normalized and embedded. Then we adopt an iterative unit-by-unit, component-by-component approach inspired by the transformers ((Singh & Mahmood, 2021)) and LLMs, such as chatGPT.

We incorporate information from surrounding units using a context window. This context ensures that the placement and arrangement of components in a given unit maintain logical consistency and align with the broader spatial and functional structure of the building.

To enhance usability of our solution, we developed a straightforward graphical user interface (GUI) using Python libraries. This GUI simplifies the process of preparing data for our AI system by providing an intuitive, user-friendly interface for handling Revit project exports instead of compiling the code by hand. With this tool, users can upload a .obj file representing a Revit project directly into the application, and then all further steps within the described pipeline are processed automatically, allowing the user to proceed directly to the component generation step.

The GUI also offers a preview functionality to ensure that users can verify their selections and the processed output. This visualization is displayed as an interactive 3D image, allowing users to explore the building's structure and layout in real-time. The interactive visualization not only provides a clear depiction of the processed data but also acts as a validation tool to ensure accuracy before proceeding to the next steps in the pipeline. Once all specifications for each unit are confirmed, the user can export a .obj file representing the generated components.

By integrating these features, the GUI bridges the gap between the technical aspects of data preparation and the practical needs of end-users. It reduces the complexity of managing exports and component selection, offering a seamless and efficient way to handle Revit project data while ensuring a clear and interactive way to visualize results.
This comprehensive pipeline, combining Revit data extraction, iterative context-aware generation, enables a robust and intelligent system for unit-based component generation and building design analysis.

III. Implementation

A. Revit data extraction implementation

The Revit program contains an add-on software called PyRevit. With this, users are capable of running their own Python scripts within Revit. However, this comes with some limitations. The Python version within Revit is a simpler version; it is unable to run a script containing other libraries, such as Pandas or Numpy. Also, running the script took significantly longer than within a standard IDE. So we opted to implement software extraction completely separately from Revit, using manually extracted object files from Revit as input.

The object files that are extracted from Revit have the following variables:

1) *usemtl*, this represents the material file that is used, in a material file the color and shade of an object is stored
2) *g*, this represented the name of the group of vertices and faces that represent one object in a mesh
3) *v*, this contains all the standard coordination vertices, represented with X,Y,Z coordinates
4) *vn*, this stands for normal vectors, these are used to add lightning and shading to an 3D object
5) *vt* this stands for texture vectors, these are used todefine 2D coordinates for mapping textures on the 3D objects
6) *f*, this contains the faces. Faces are a triple of coordinates that make a small triangle, combining all of the faces will result in one 3D object.

Since for this project, the only thing of interest is the locational data, which makes the shade, lightning and the texture of the objects are unnecessary. Storing the material file, normal vertices and texture vertices is thus redundant. So when the object file is read for the first time, the program makes sure to delete all of these instances, drastically reducing the amount of data that needs to be stored.

When the object file of the building is being read, while the unnecessary information as just described gets removed, the rest is immediately sorted into respective groups. Not every object in building-mesh needs to be processed by the software. Walls, floors and ceilings are the only things that matter when placing MEP components.

The software has a set of keywords that will filter out the unwanted objects. So the needed objects are accepted, and a dictionary is made to store the objects name, vertices and faces, this will then be added to a list of all discovered groups. This reduced list of groups for each unit, will be put together and written again into a object file, called *mesh.obj*, since this represents the 3D structural data of the unit that is needed by the AI models. One of these units can be seen in the appendix figure 14.

Given the list of the groups, the distinct units can be calculated with their minimum and maximum coordinates. The units' position will be compared to each-other to find out which units are adjacent to which units. If a unit doesn't have an adjacent unit on a side, the size of the unit will be expand to that side with a meter. This results in the program being able to detect certain MEP components that are on the outside of a unit. Each unit will get their own dictionary with their unit id, minimum coordinates, maximum coordinates and their list of adjacent units. Also, the dictionary will contain an empty list for every MEP type, this is where the components will be

stored that are in that unit. This process will deliver all the structural information about every distinct unit.

Obviously, the same kind of process needs to happen with the MEP components. Again, when the component object file is being read, it is split into groups while unnecessary data gets removed. Before the components can be linked to an unit, another step has to be taken. Some type of components are built up from multiple smaller components. For example, a standard outlet is built up from 8 smaller components. So these 8 smaller components are merged together, into one group, making one big composite component. This is done for all the components that this is applicable for. A Json file has been made where the name of the component is stored, together with the amount of components needed to make one big composite component. This makes the amount of components per unit drastically decrease. Afterwards the components are linked to a unit and the program checks if the middle of the component is within which unit. The component will be added to the dictionary of the unit.

In the end of this pipeline, per unit, 2 files will be made; one file, *mesh.obj*, containing the meshes of the unit and the other file, *data.json*, containing the units information dictionary. These will be handed towards the AI model for training purposes.

B. Data Normalization and Embedding

To address the problem of generating components using a neural network, the data needs to be transformed into a numerical form that unambiguously defines the input. Furthermore, it is preferable that the input vector values are in the range of $< 0, 1 >$, allowing the neural network bias nodes to contribute to the calculations even in the initial stages of the inference.

Our neural networks take three types of inputs that are concatenated:

1) placed components list
2) mesh
3) component to be placed (only in case of location predictor NN)

1) Embedding of placed components list

The data we're considering contains numerous components of various types; as such, it is of vital importance to include all this in a possible embedding. Initially, we included this data as a flattened matrix of size NxM, where N was the number of already placed components in the unit and M was equal to the $3 + c$, where 3 is the 3 degrees of freedom representing the component's position and c is the number of all the component types which appeared in the training dataset. In such a matrix the component type was specified using a one-hot vector of length c. This vector was then padded with zeros to a predecided, during training, size.

Nevertheless, this approach suffered from a very serious problem. As the resulting matrix was sparse, due to only 4 out of every tens or even hundreds of values containing non-zero values, this approach was not scalable. The embedding we developed to address this problem was to create a correspondence between each set of 3 subsequent values in the vector and a component type. Resulting embedding vector efficiently utilizes its space to encode the components at the cost of decreasing the flexibility of the embedding process, possibly creating issues when applied on unknown datasets.

The number of the sets allocated to each component type would be equal to the sum of number of times that component type appeared at most across all the units in the training dataset and the offset value, which we have set to be 2 to allow the usage of our neural network on the datasets containing more components per unit than seen before.

In order to normalize the values, the location of the unit is deducted from the location of each component and the resulting value is divided by 7000. This value is derived from the fact that the largest of the dimensions of the units, which we assume are standardized is 7000 [mm]. As such regardless of how the units are rotated the values of the locations of the components are within the range $< 0, 1 >$.

2) Mesh

The implemented embedding process for the meshes is relatively straightforward. Once the mesh is loaded from the .obj file, n points are sampled from it. The coordinates of these points are then normalized in an exactly same manner as the coordinates of the components. The normalized values of the coordinates of the points are then embedded into a vector of length 3xn.

As sampling randomly shuffles the order of the points, an embedding acquired in this way cannot be expected to have a strict internal structure. Possibly making the application of the Convolutional Neural Networks, which view small "local" parts of the embedding, less effective. In order to mitigate this issue and impose some level of structure we sort all the points by their "x" coordinate.

3) component to be placed

In order a embed a component type of the component we want to place, we construct a one-hot vector v of length c, where c is the number of different component types in the training dataset and set v[i] = 1, where i is the index of the given component type as determined during the training process. All the remaining values of the vector are set to 0.

C. AI Model

Using PyTorch ((Paszke et al., 2019)), we have developed two AI models that operate iteratively and sequentially in order to generate a solution:

1) Type Predictor
2) Location Predictor

1) Type Predictor

The type prediction model is a multi-layer neural network of following specifications:

- **Input:**
  1) Already placed components,
  2) unit mesh,
  3) components placed in the surrounding units,
  4) meshes of the surrounding units.

- **Output:** A vector v of length c, where v[i] represents the probability that the component with component type of index i should be placed in a unit.

As can be seen from the description of our output, we didn't formulate the problem as a classification problem, as would normally be expected, but as a regression problem instead. As such, we don't normalize the output using the Softmax function, as is common with classification problems.

We have made such a decision as the predicted likelihoods of component types are not necessarily exclusive, i.e. the fact that component A should be placed in the unit, does not necessarily mean that the component B shouldn't be placed.

Ultimately, the component type that is chosen to be placed next is the one with the highest likelihood.

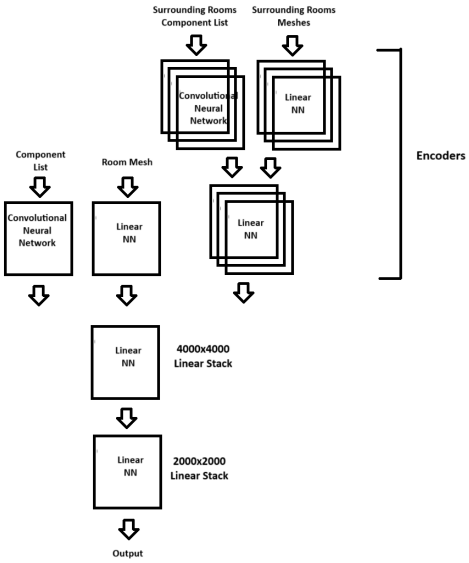

Fig. 5. Architecture of one of the type predictor models

2) Neural Network for Location Prediction

The location prediction model builds upon the architecture of the type predictor but incorporates the encoder of an additional input type, allowing for to-be-placed component specification.

- **Input:**
  1) Already placed components,
  2) unit mesh,
  3) one-hot vector, representing to-be-placed component,
  4) components placed in the surrounding units,
  5) meshes of the surrounding units.
- **Output:** A vector of length 3 representing the coordinates the component should be placed on (x, y, z).

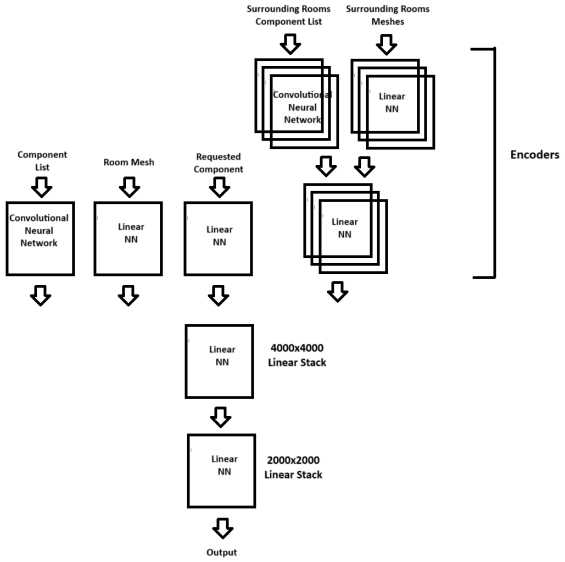

Fig. 6. Architecture of one of the location predictor models

3) Neural Network Internal Architecture

The neural network was designed to facilitate a highly scalable architecture. As such during the inference stage each of the input types is first considered separately. A smaller encoder is trained for each input type to encode the data into a latent space vector, facilitating a level of data compression within the neural network.

To encode the surrounding units, first, each input type for each unit is run through a corresponding encoder. Then, the latent space vectors are concatenated, resulting in the latent space vector for the surrounding units.

Once the latent space vectors of all inputs are acquired, the vectors are concatenated, and a stack of linear layers is applied. Afterwards, a stack of smaller linear layers is applied again, adjusted so that it returns data in the correct format.

Our models incorporate Batch Normalization, described in (Ioffe & Szegedy, 2015), which helps in minimizing the covariance shift negatively affecting the learning process, allowing for faster and more predictable training.

Our best models also use Convolutional Neural Networks, which allow us to take advantage of the internal structure of our embedding of already-placed components. These CNNs proved very useful in extracting spatial relationships among embedded component positions. After these convolutions, we apply a flattening layer, feeding it into fully connected linear layers.

To ensure that our neural networks do not suffer from dying neurons, we're using the leaky ReLu as our activation function.

D. AI Model Training Framework

Training framework for the neural networks might be just as important as the internal neural network architecture and the input embedding. The choice of an appropriate loss function might be a deciding factor on whether the neural network will memorize the training dataset or learn the general patterns, it

might also be a deciding factor between a steady convergence and stopping at a high local minimum.

The training framework we developed takes these considerations into account.

1) input-output pair generation

During a single Epoch, the units in the training datasets are merged into single set of units. Then the number of cumulative component counts for each unit is computed.

The highest value m is chosen as a reference for the number of batches that will be generated.

The generated number of batches b:

$$b = floor(k * m)$$

, where k is a value between 0 and 1, which leverages the epoch length and the amount of progress to be achieved (and permutations considered) within a single epoch. The value we used is $k = 0.3$ for training dataset, and $k = 0.1$ for the test and verification datasets.

input generation: Each batch contains examples of all the units within the training dataset, with a random number of already placed components, specific components were also randomly chosen. This approach allows us to take into account a significant number of possible already-placed component lists, minimizing risk of overfitting.

For each input-output pair, a new set of points is sampled from the loaded mesh to ensure that the neural network doesn't memorize the units by their meshes but can instead consider a large set of possible 3D geometries.

To generate a to-be-placed component type for the location predictor, the generated already-placed component list is compared with the list in the training dataset and a random component type among those of which instances are more numerous in the list in the training dataset is chosen.

output generation:

For the location predictor, the output is a list of all the coordinates the to-be-placed component type is located in and which are not already present in the generated already-placed component list, which is part of input.

For the type predictor, the output is a vector v of length c, where c is the number of all component types the neural network was trained on. $v[i] = 1$ if the corresponding component type's instances are more numerous in the training dataset than in the generated already-placed component list, which is part of input, else $v[i] = 0$.

loss functions:

For the location predictor, the loss is $min(RMSE(output, position_i))$, where $position_i$ corresponds to the i_{th} position in the reference output. This approach allows for a dynamic adjustment of the positions proposed by the neural network, guaranteeing that all possible solutions would be accepted.

For the location predictor, the loss is $RMSE(output, ref\_output)$.

Optimizer: Optimizer used is the Stochastic Gradient Descent, providing a predictable traversal of the solution space. The parameter values, which were used, are:

- learning rate = $10e^{-5}$
- momentum = 0.1

Test and Verification loops: The test and verification loops were constructed analogously, excluding the backpropagation step, to test the model's performance during training.

E. Autonomous component generation

The application of our neural networks is relatively easy. Given an arbitrary threshold t between 0 and 1.

1) apply the type predictor on a unit
2) if none of the values in the type predictor's output is above t, then stop generation
3) apply the location predictor on a unit with the component corresponding to the largest value in the type predictor's output as a to-be-placed component
4) add the to-be-placed component with the generated location to the unit
5) return to step 1)

F. Metric for measurement of the accumulated error

Given a unit for which we have a reference solution and have a generated solution, two methods were developed to ascertain the accumulated error.

1) Number of generated components without corresponding component in the reference solution

This metric is very straightforward the number of generated components and reference components per type per unit is computed and the two sets are compared.

For example, given component type A:

- If there are 2 components of this type in the reference solution and 1 was generated, the number of components without a corresponding one is 0.
- If there are 2 components of this type in the reference solution and 2 were generated, the number of components without a corresponding one is 0.
- If there are 2 components of this type in the reference solution and 4 were generated, the number of components without a corresponding one is 2.

2) The minimum-distance maximal matching between the generated components and the reference components

To compute this metric, a distance matrix is first constructed between the generated components and the reference components.

Afterwards, the networkx library ((Hagberg, Schult, & Swart, 2008)) is used to phrase the problem as a matching problem in a bipartite graph.

This metric provides a good overview of the actual accumulated placement error by punishing the algorithm for concentrated placement of components if the reference components are not concentrated as well.

G. Expert System for Location Prediction

The expert system introduces a decision tree regressor to predict the placement of components within a unit. The trees operate on a set of features derived from the JSON files representing architectural models from four buildings. These

features include variance in the x, y, and z coordinates, distances to the nearest walls, and the dimensions of the components. Additionally, each unit was divided into eight equal parts, twice along x, y, and z axes, and the subregion where each component is located was encoded as a categorical feature. Feature importance analysis revealed that unit subdivisions and component dimensions were the most significant predictors for accurate placement. Conversely, coordinate variance contributed less due to the minimal variation observed for many components.

After feature extraction, the dataset was divided into training (80%) and testing (20%) subsets to ensure robust model evaluation. Three separate decision tree regressors were trained to predict the minimum x, y, and z coordinates of a component's placement. These predictions correspond to the minimum corner of the component's bounding box and were combined with the component's dimensions to calculate the bounding box's placement within the unit. To ensure valid placement, the system uses a collision detection mechanism. For each component, the algorithm checks for overlaps with previously placed components. If a collision was detected, the placement was adjusted incrementally using a brute-force approach until the component is successfully placed.

The hyper parameters of the decision trees were optimized to balance performance metrics. The final configuration set the maximum depth to 10, the minimum sample split to 2 and the minimum samples leaf to 1. This method provides a deterministic and more interpretable baseline for component placement, providing a comparison with more advanced neural network models we developed.

H. Object file reintegration

The output generated by the AI is a JSON file containing a list of 2 things; the name of the generated component combined with their generated minimum location. Now this data should be reintegrated back into an object file to show the end result. This is done by having a stored mesh and their respective minimal location of every distinct component saved. The generated minimal location will be matched by name with a saved component.The next step will be compared to the stored minimal location with the output minimal location, their difference represent the offset of the new component. In order to place the new component in the correct position, the difference will be added to all the vertices of that group. This results in a correctly placed component per unit, which then can be combined in a final object mesh contain the whole generated MEP for a building, as can be seen in the appendix, figure 15 and figure 16. The red components represent the generated components.

I. GUI

The GUI created for this project uses the TKinter and PyVista libraries in Python to ensure a user friendly environment. An environment in which the user can input the building obj file manually extracted from Revit, select the output file path, enter the building name, view the dynamically created floor plan, select the quantity of components per unit and view the unit with its component in a 3D-viewer. Most of these of function were made with the TKinter package, which the most used standard in Graphic User Interfaces for python. The 3D-viewer was made using PyVista which a 3D plot and mesh type package from C++ put in a python wrapper.

J. Evaluation Methodology

The evaluation framework we developed for this project compares our generated building models with a solution model to assess their accuracy. We aim to evaluate the positional accuracy of component placement, whether the right components were placed and if they were placed correctly. To achieve this we make use of the metrics below.

IV. Experiments

A. Neural Network experiments

We conducted experiments to evaluate the performance of our neural network models under various configurations and enhancements. Throughout the experiments, we collected training, testing, and verification losses. These metrics allowed us to comprehensively assess the effectiveness of the proposed modifications.

The training and test datasets were constructed from three datasets, which are named "gebouWe", "opvang" and "paviljoen". The training dataset used 70% of the units in the datasets, and the test dataset used the remaining 30% of the units.

The verification dataset was constructed from "rotterdam" building.

Specifically, we compared the baseline performance of our simplest neural network implementation with several enhanced versions. These enhancements included the incorporation of a context window to provide additional information, the integration of convolutional neural networks (CNNs) to leverage spatial hierarchies in the component list data and unit meshes. By systematically analyzing these configurations, we aimed to determine the impact of each improvement on the overall accuracy, stability, and generalization of the model.

Additionally, we have autonomously generated the entire "paviljoen" building 10 times with threshold of 0.9 and computed the average number of generated components without corresponding component in the reference solution and the minimum-distance maximal matching between the generated components and the reference components. The neural networks used for this are the further trained (for around 200-250 epochs) versions of best performing architectures.

B. Expert system experiments

We experimented on the tuning of the model. We got that the model predicts the best when these parameters are set for the decision tree regressor, MAX DEPTH = 10, MIN SAMPLES SPLIT = 2, MIN SAMPLES LEAF = 1.

We conducted experiments to evaluate the performance of our expert system for placing components. The system's performance was evaluated using the following metrics: Mean

Absolute Error (MAE) for X, Y, and Z coordinates. Root Mean Square Error (RMSE) for X, Y, and Z coordinates. A Euclidean MAE and Euclidean RMSE were calculated to provide a single aggregated measure of placement error across all dimensions. This was done for the placing of each component in all units of buildings this for all the units in buildings. We then saved each unit's measurements. The evaluation process was conducted for all units within multiple buildings (gebouwE, opvang, paviljoen, rotterdam). For each unit, the system predicted the placement of all components and compared the predicted placements against ground truth data from the JSON files. Metrics were calculated for each unit. Then the average metrics Euclidean MAE and Euclidean RMSE were calculated for each building to summarize overall placing performance. The

Detailed metrics for each unit are presented in a CSV file, allowing for analysis of individual unit performance. Average Building-Level Performance was computed for each building, demonstrating the system's ability to generalize across different spatial layouts.

V. Results

The results of the experiments reveal significant differences in model performance based on the configurations tested.

A. Loss Metrics

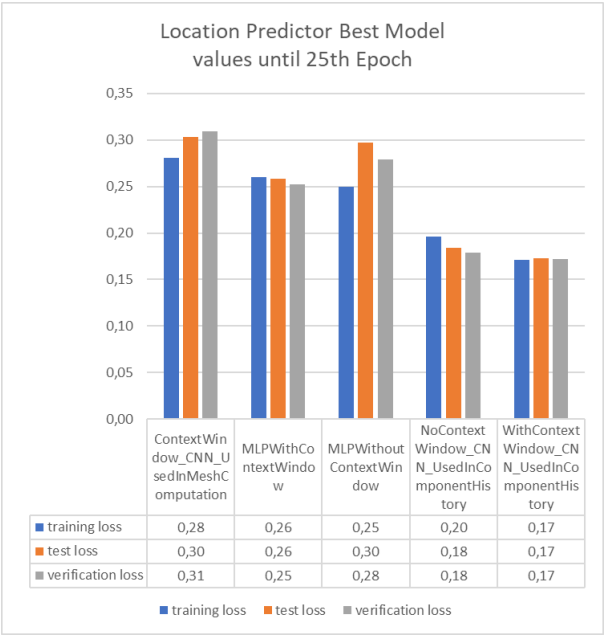

Fig. 7. The architectures that utilize the CNNs seem to perform best

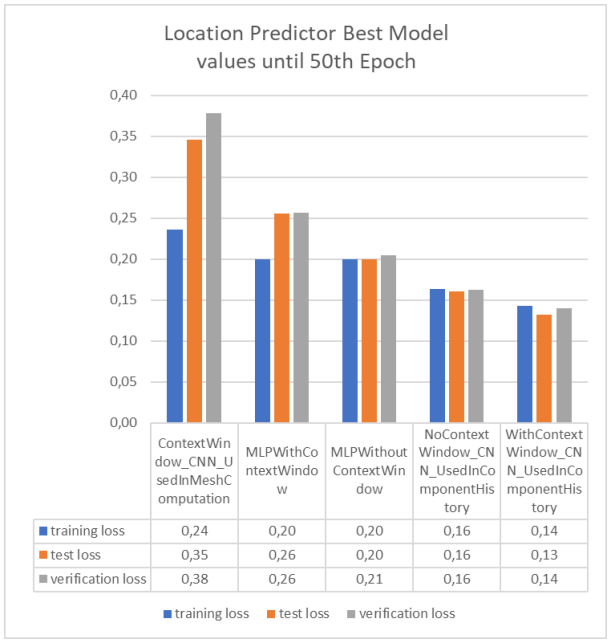

Fig. 8. The architectures that utilize the CNNs seem to perform best, the linear neural networks seem to work better with the implemented mesh embedding

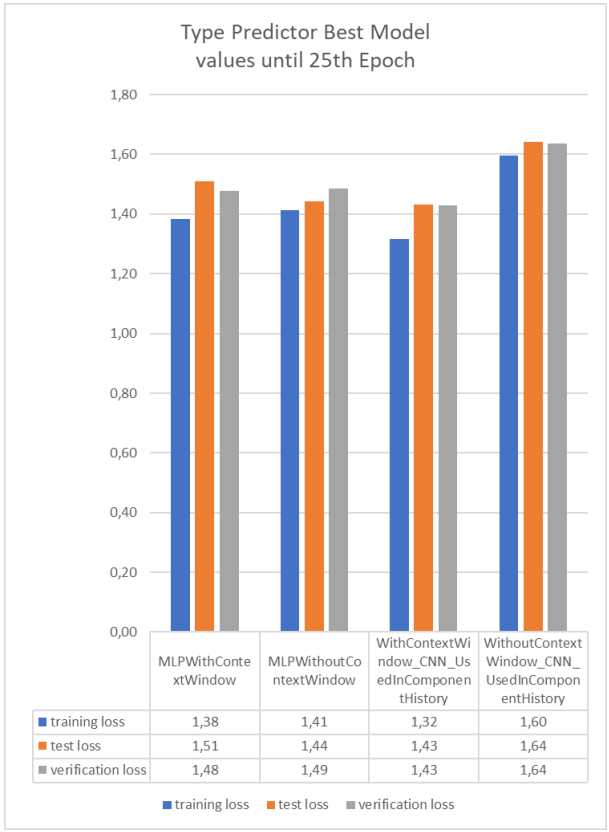

Fig. 9. The architectures utilizing context window seem to have a slight advantage over the other architectures

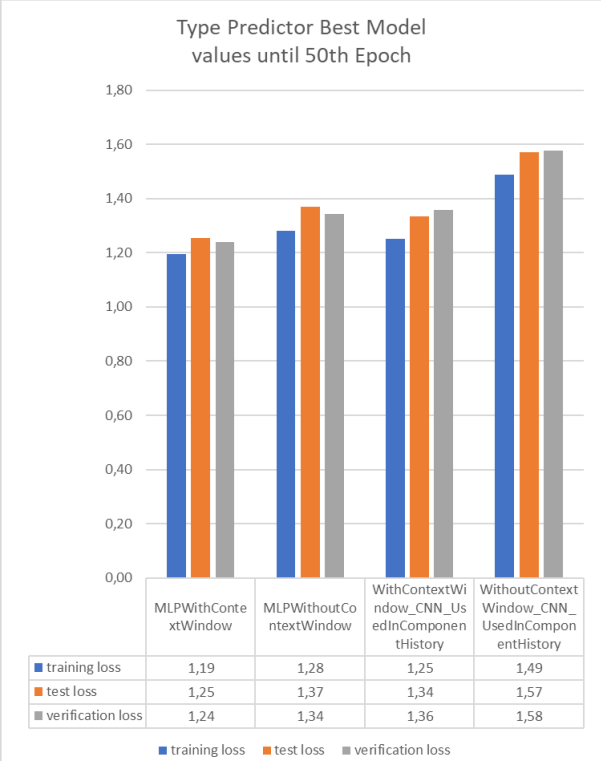

Fig. 10. The MLP seems to have best results for the type prediction taks

B. Accumulated error metrics

	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10	
sum of mismatched components	101	127	95	107	103	128	164	177	102	90	122,6667
generated component count	403	421	361	402	377	436	473	478	411	362	418
incorrect ratio	0,25062	0,301663	0,263158	0,266169	0,27321	0,293578	0,346723	0,370293	0,248175	0,248619	0,290399
mean placement error	1,147893	1,103457	1,132028	1,128173	1,134612	1,109938	1,137405	1,102931	1,144893	1,12706	1,126815

Fig. 11. Results of autonomous component generation with threshold of 0.9

C. Expert System Performance

We observed that the tree for the z coordinate is much more accurate than the ones for the x and y coordinates. The biggest error for z is 520mm, while for x and y it is 2070mm and 2000mm.

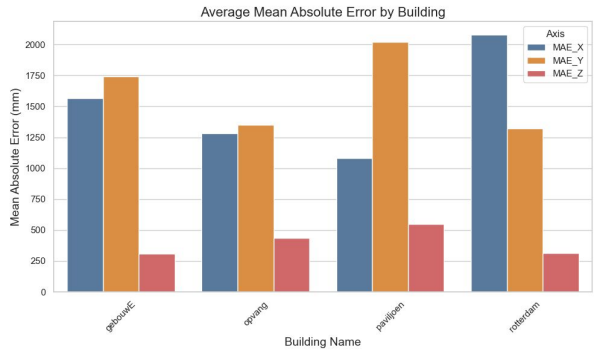

Fig. 12. MAE for predictions of each individual axis

Furthermore, when comparing the Euclidian distance of components to the ones in the test data, we get a mean absolute error per building between 1800-2500mm. For comparison an agent that places components randomly performs with MAE 4000mm.

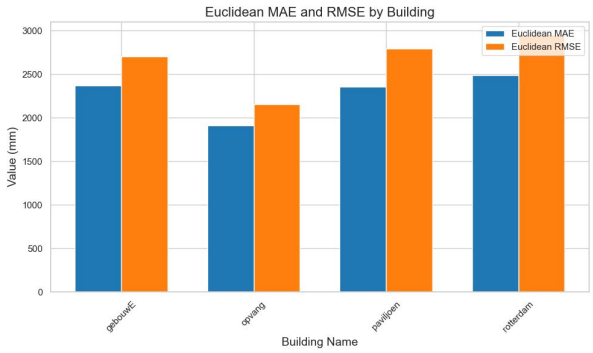

Fig. 13. MAE for predictions of each individual axis

The expert system has an average error per building of 2279.18559mm. For comparison the AI driven agent performs with an average error of 1701.42839mm, while a random agent with randomized placements has 4000mm.

VI. Discussion

The training, test, and verification losses clearly show model performance. All configurations show similar values across these metrics, especially during later epochs. This consistency suggests that the models performed comparably across the different datasets, indicating stable learning outcomes and the implemented approach's good generalizability.

A. Impact of the context window

The use of a context window plays a pivotal role in enhancing model performance. Models incorporating context windows consistently outperform those without. This highlights the importance of incorporating surrounding unit data to provide a much better contextual understanding during prediction.

B. Importance of the CNNs

The integration of CNNs in the component list significantly impacts model performance in location prediction. When CNNs are used to process component history with a context window, the model achieves the lowest training loss (0.14), test loss (0.13), and verification loss (0.14). This indicates that the combination of CNN-based feature extraction and context-aware design offers the most effective solution for predictive accuracy.

In contrast, CNNs applied to unit mesh analysis with a context window result in higher losses across all metrics (training: 0.24, test: 0.35, verification: 0.38). This suggests that while CNNs are capable of extracting geometric features, they seem unable to take advantage of the internal structure of the implemented embedding of the meshes.

Surprisingly, applying CNNs to the component list doesn't improve the results and might instead negatively affect them.

Although we lack data to say with certainty, a possible hypotheses might be that during type prediction, the inter-component type relationships are more important for the model than the intra-component type relationships. The CNN, which looks only at a narrow view of the components, might not be able to guarantee this, as well as fully connected layers.

C. Performence upon autonomous application

Although performed on the training dataset, the results of the computed accumulated error metrics show a very good performance in type prediction of around 70% with the generated component count of around 400. While the mean placement error is within 1.1 meter leaving some area for improvement. This performance is sufficient to show promise in the developed neural networks, especially that the neural networks that achieved this were trained on a personal computer for just a few hours.

D. Expert systems

Our experiments tell us that a rule-based approach while not as accurate as the AI, is still worthwhile since it performs relatively well and much better than random placement. It serves as a good baseline for the neural network and shows that our approach is feasible and could be enhanced even further with future improvements.

The decision trees in our expert system generate rules based on the position of components in the test data. Future research teams could analyze the specific functionality of components and the manners of their placement to manually create additional rules. If the task expands to include different categories of components (for example, plumbing components), the trees could use the knowledge gained from this additional attribute.

VII. Conclusion

In this proof-of-concept study, we developed and tested AI-driven and expert systems methods to automatically generate MEP components in buildings. The best performing method we had was our CNN-based neural network, particularity when merged with a context window referencing the surrounding units. The implementation of the context window demonstrated consistently lower placement errors compared to both the baseline linear methods and the expert system decision trees. The expert system, while achieving lower accuracy, provided an interpretable and deterministic alternative, making it a valuable baseline.

Our results show that including CNNs in the neural network architecture might significantly improve performance. However, this requires prior consideration of the internal structure present in the embedding and whether the CNNs are capable of fully utilizing it.

More efficient or structured embeddings of the unit meshes and less restrictive embedding of the component lists should be considered during future work. As at the moment the spatial limitations of the model limit the possible scope of its application and its flexibility.

The results prove the feasibility and potential time savings of automated MEP modeling. By embedding existing component layouts, the neural networks effectively learned spatial relationships, recognizing patterns in units and component embeddings well. These initial findings highlight how contextual information, such as adjacent units, can significantly improve predictive accuracy for both component type selection and placement.

Ultimately, our pipeline presents a scalable solution that can further benefit from evolving deep learning techniques and growing modular construction demands, thus reducing human modeling effort and construction lead times in modern architectural practices.

Dita Hořínková. (2021). *Advantages and disadvantages of modular construction, including environmental impacts.* Brno University of Technology, Faculty of Civil Engineering, Institute of Technology, Mechanization and Construction Management, Veveří 331/95, 602 00 Brno, Czech Republic: IOP Publishing Ltd. Retrieved from https://iopscience.iop.org/article/10.1088/1757-899X/1203/3/032002/meta (DOI 10.1088/1757-899X/1203/3/032002)

Hagberg, A. A., Schult, D. A., & Swart, P. J. (2008). Exploring network structure, dynamics, and function using networkx. In G. Varoquaux, T. Vaught, & J. Millman (Eds.), *Proceedings of the 7th python in science conference* (p. 11 - 15). Pasadena, CA USA.

Ioffe, S., & Szegedy, C. (2015, 2). *Batch normalization: Accelerating deep network training by reducing internal covariate shift.* Retrieved from https://arxiv.org/abs/1502.03167

Kazeem KO, A. L., Olawumi TO. (2024). Integration of building services in modular construction: A prisma approach. *Applied Sciences*. Retrieved from https://www.mdpi.com/2076-3417/14/10/4151

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... Chintala, S. (2019, 1). PyTorch: An Imperative Style, High-Performance Deep Learning Library. *arXiv (Cornell University)*, *32*, 8026–8037. Retrieved from https://arxiv.org/pdf/1912.01703.pdf

Ravi, N., Reizenstein, J., Novotny, D., Gordon, T., Lo, W.-Y., Johnson, J., & Gkioxari, G. (2020, 7). *Accelerating 3D Deep Learning with PyTorch3D.* Retrieved from https://arxiv.org/abs/2007.08501

Singh, S., & Mahmood, A. (2021, 1). The NLP Cookbook: Modern Recipes for Transformer Based Deep Learning Architectures. *IEEE Access*, *9*, 68675–68702. Retrieved from https://doi.org/10.1109/access.2021.3077350 doi: 10.1109/access.2021.3077350

Teo, Y. J. H. A. H. Y. S. C. M. Z. L. C. J. . C. K. H., Y. H. (2022). Enhancing the mep coordination process with bim technology and management strategies. *sensors*. Retrieved from https://doi.org/10.3390/s22134936

A. MEP models

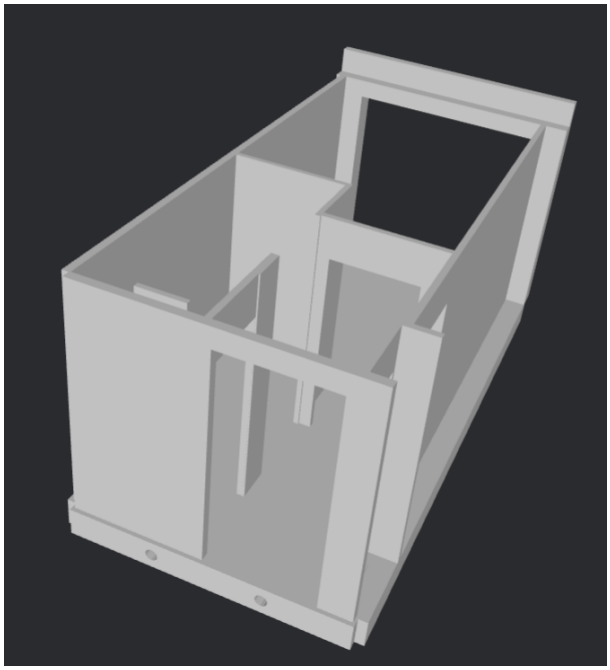

Fig. 14. Mesh of one unit

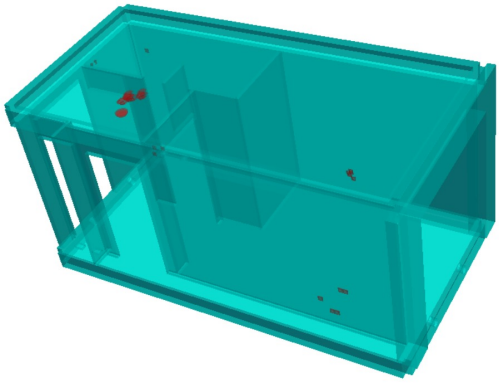

Fig. 15. MEP solution side view

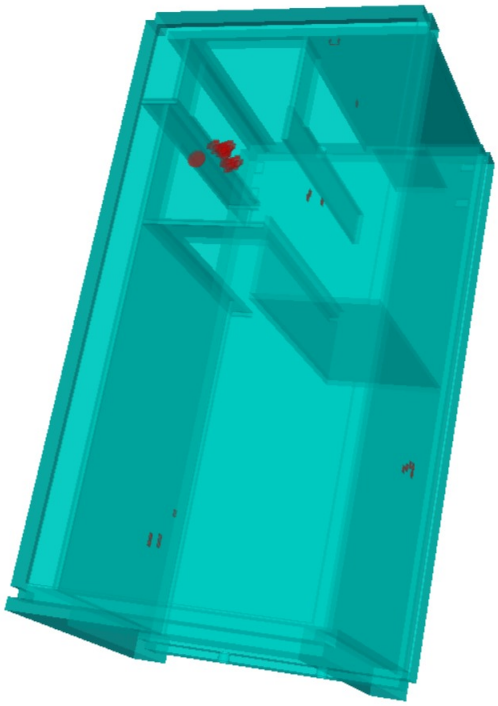

Fig. 16. MEP solution top view