ASME Journal of Mechanical Design
Online journal at:
https://asmedigitalcollection.asme.org/mechanicaldesign

Large Language Models for Computer-Aided Design Fine Tuned: Dataset and Experiments

Yuewan Sun
Walker Department of Mechanical Engineering,
University of Texas at Austin,
Austin, TX 78712
e-mail: yuewansun@utexas.edu

Xingang Li
Walker Department of Mechanical Engineering,
University of Texas at Austin,
Austin, TX 78712
e-mail: xingang.li@utexas.edu

Zhenghui Sha[1]
Walker Department of Mechanical Engineering,
University of Texas at Austin,
Austin, TX 78712
e-mail: zsha@austin.utexas.edu

Despite the power of large language models (LLMs) in various cross-modal generation tasks, their ability to generate 3D computer-aided design (CAD) models from text remains underexplored due to the scarcity of suitable datasets. Additionally, there is a lack of multimodal CAD datasets that include both reconstruction parameters and text descriptions, which are essential for the quantitative evaluation of the CAD generation capabilities of multimodal LLMs. To address these challenges, we developed a dataset of CAD models, sketches, and image data for representative mechanical components such as gears, shafts, and springs, along with natural language descriptions collected via Amazon Mechanical Turk. By using CAD programs as a bridge, we facilitate the conversion of textual output from LLMs into precise 3D CAD designs. To enhance the text-to-CAD generation capabilities of GPT models and demonstrate the utility of our dataset, we developed a pipeline to generate fine-tuning training data for GPT-3.5. We fine-tuned four GPT-3.5 models with various data sampling strategies based on the length of a CAD program. We evaluated these models using parsing rate and intersection over union (IoU) metrics, comparing their performance to that of GPT-4 without fine-tuning. The new knowledge gained from the comparative study on the four different fine-tuned models provided us with guidance on the selection of sampling strategies to build training datasets in fine-tuning practices of LLMs for text-to-CAD generation, considering the trade-off between part complexity, model performance, and cost. [DOI: 10.1115/1.4067713]

Keywords: computer-aided design, multimodal data, large language model (LLM), fine-tuning LLMs, artificial intelligence, data-driven design, generative design

1 Introduction

Recent advancements achieved by the generative pretrained transformer (GPT) series of large language models (LLMs) have shown great potential for artificial intelligence (AI) to interact with the world [1]. These models exhibit remarkable proficiency in “understanding” the nuances of human language and thus can better interact with humans [2].

More recently, the capability of LLMs in processing natural language has extended their applications in design research, especially conceptual design [3–5]. These studies typically use textual information to represent design requirements for the generation of design concepts with LLM. Moreover, incorporating multimodal datasets, such as combinations of textual and visual data, can enhance LLMs’ ability to understand and interpret design intent more effectively.

In conceptual design, many design decisions are made based on the 3D shape of a product [6]. So, one of the key motivations of applying multimodal LLMs to conceptual design is to enable them to generate computer-aided design (CAD) of 3D shapes (referred to as MLLM4CAD (multimodal large language models for computer-aided design) hereafter). However, to our knowledge, while one study conducted qualitative evaluations of MLLM4CAD using the ChatGPT interface [7], no quantitative evaluation has been performed to assess the efficacy of MLLM in 3D CAD generation.

In our previous work [8], we developed a multimodal CAD dataset and explored the zero-shot capabilities of GPT-4 in generating CAD programs with different combinations of design modalities. We also introduced a debugging process to improve performance in the generation of CAD programs for the GPT-4 and GPT-4V models. However, there remains room for further improvement. To address this, LLM fine-tuning has been identified as the most promising approach [9]. However, to the best of our knowledge, there is a lack of existing datasets for LLM fine-tuning in 3D CAD generation tasks. To overcome this, we expanded our multimodal CAD dataset (including text descriptions and 2D images and sketches of mechanical components) by incorporating fine-tuning data and subsequently used this enriched dataset to develop a fine-tuned GPT model in support of LLM4CAD tasks.[2] With the fine-tuned GPT model, we aim to accelerate the design

[1] Corresponding author.

Contributed by the Design Automation Committee of ASME for publication in the Journal of Mechanical Design. Manuscript received August 10, 2024; final manuscript received January 2, 2025; published online February 27, 2025. Assoc. Editor: Faez Ahmed.

[2] At the time of this study, only the GPT-3.5 model was available for fine-tuning. So, we adopted the “gpt-3.5-turbo-0125” model, but it could only support text as input. Therefore, this article focuses on fine-tuned LLMs in text-to-CAD model generation.

process of relatively simpler geometries yet with certain customized features, helping to save time and reduce costs during the initial stages of design.

2 Background Knowledge

In this section, we introduce the background knowledge of design modalities, LLMs, and transfer learning.

2.1 Design Modalities and Large Language Models.

Design modalities refer to the use of various data modalities during the process of product design and development [10,11]. Common design modalities in early design stages, e.g., conceptual design, can be categorized into three primary categories: (1) textual data (e.g., natural language description). This modality encompasses customer needs and design requirements that are typically represented and encoded in written or spoken language. (2) 2D visual data (e.g., sketches, drawings, and images). This modality enables designers to quickly capture and visualize preferences and ideas during the conceptual design stage. (3) 3D visual data (e.g., CAD models): Low-fidelity design concepts and prototypes in the conceptual design stage are commonly represented through 3D digital shapes, providing a better approximation and representation of the final design artifact [10]. The integration of these modalities ensures a richer and more precise representation of design ideas, highlighting the interconnected nature of data in the design process.

The importance of multimodal data lies in its ability to bridge diverse forms of representation, fostering a deeper understanding of complex design challenges. By combining textual, 2D visual, and 3D visual data, designers can explore a broader design space and communicate ideas more effectively. Recent advancements in AI have introduced opportunities to leverage such data through models capable of processing and learning from multiple modalities simultaneously. These advancements not only enhance the fidelity of design representations but also open new avenues for automation and optimization in conceptual design workflows. Multimodal datasets thus play a critical role in improving the accuracy and efficiency of design exploration and decision-making.

The evolution in this area involves multimodal large language models (MLMMs) that seek to improve the abilities of LLMs by combining multisensory skills (e.g., sound, vision, video), leading to greater intelligence [12]. Textual and visual information are two prominent modalities in our everyday life, and most problems require modeling both to generate satisfactory outcomes [13]. As a result, many MLMMs start by extending the vision capability in addition to the textual capability of LLMs [12,13]. Prior research generally adopts one of two strategies: leveraging a vision-language model to transform visual data into text-based descriptions that LLMs can process [14–16] or fine-tuning a vision encoder to integrate with a frozen pretrained LLM [17–19]. For example, the evolution of OpenAI's GPT architecture from text-only versions to multimodal successors such as GPT-4 Vision shows significant progress with opportunities to incorporate multiple modalities beyond text [20].

2.2 Zero-Shot Learning and Fine-Tuning.

Zero-shot learning is a technique in deep learning aiming to apply the trained models to tasks or classes it has not encountered (i.e., unseen data) during training [21,22]. In the context of LLMs, the zero-shot learning capability refers to the model's ability to generalize from its pre-trained knowledge base. For instance, GPT-3 can perform tasks such as translation without explicit task-specific training, relying solely on the contextual understanding developed during pretraining [1]. This capability is particularly valuable in scenarios where labeled data for new tasks are unavailable or scarce.

Fine-tuning is a specific form of transfer learning that involves further training of a pre-trained model on a smaller, task-specific dataset. This process adjusts the model's parameters to better fit the new task. Fine-tuning can be applied to the entire model or selectively to certain layers, depending on the amount of available data and the complexity of the task. In the context of LLMs, fine-tuning can significantly enhance performance on specific tasks such as sentiment analysis or question answering, by tailoring the model's general linguistic knowledge to particular requirements of a task [23–25].

Zero-shot learning enables LLMs to perform tasks without task-specific training, while fine-tuning tailors pretrained models to specific tasks using smaller datasets. Together, these methods demonstrate the flexibility of LLMs in adapting to a wide range of applications, from generalizing to unseen tasks to excelling in specialized domains.

3 Literature Review

In this section, we review the relevant literature on LLMs, MLLMs, CAD model representation, and 3D shape datasets in engineering design research.

3.1 Engineering Design Using Large Language Models and MLLMs.

We review the relevant literature on engineering design using LLMs and MLLMs based on the input design modalities. The input modality refers to how information is fed into a model.

Regarding the input modality, most existing research has focused on feeding information into LLMs using text-based prompts [4,26–32]. Text-based prompts allow designers to directly communicate their design requirements and objectives to a model. However, design intent can be challenging to express through text alone, as design elements typically include structural and layout relationships among components with specific shapes. Visual inputs, such as sketches or images often used in conceptual design, can complement textual information to represent these relationships. Therefore, multimodal approaches that integrate both textual and visual inputs hold promise to generate more coherent design responses from LLMs.

These studies [26,28,33,34] provide valuable insights into the potential applications of LLMs and MLLMs for generating 3D shapes through CAD program synthesis. However, the limited number of examples used for the qualitative assessment restricts a comprehensive understanding of how effectively LLMs and MLLMs can be utilized for 3D shape synthesis. Our previous work [8] quantitatively evaluated GPT-4 and GPT-4V's capability in generating 3D shapes through CAD program synthesis using a multimodal CAD dataset that includes textual descriptions, sketches, images, and 3D shapes of mechanical components. While the results demonstrate the potential of these models for generating 3D shapes, there remains room for improvement, even with a proposed debugger to iteratively revise the synthesized CAD program codes.

3.2 Computer-Aided Design Model Representation.

In the context of CAD model representation, since LLMs currently cannot directly generate 3D objects (e.g., meshes, voxels, or boundary representations), an appropriate intermediate representation must be used. Two commonly adopted CAD representations that LLMs can generate include domain-specific languages (DSLs) and CAD programming languages.

The first representation, DSLs for CAD sequences, such as ShapeAssembly [35], introduces a "shape assembly language" for constructing 3D shape structures. These structures are created by declaring cuboid part proxies and attaching them hierarchically and symmetrically. While procedural programs using ShapeAssembly can easily generate related families of shapes, the learning curve for LLMs is steep due to the self-defined nature of the language [36]. Even with fine-tuning, LLMs struggle to debug such programs independently without extensive domain-specific pretraining.

Another advantage of DSLs is their ability to be seamlessly embedded into CAD sequence embeddings, as demonstrated in methods like DeepCAD [37] and SkexGen [38]. These methods encode CAD sequences into compact numerical vectors, enhancing their efficiency for machine learning applications. However, numerical embeddings face significant interpretability challenges. The transformation into vectors obscures the direct meaning of the design, making it difficult to trace back to the original CAD intent. Moreover, this process can result in the loss of intricate details and nuances in CAD designs, potentially compromising accuracy in high-precision tasks.

The second representation involves CAD programming languages, such as CADQuery [26] and OpenSCAD [33], which serve as a medium for CAD models. Since LLMs are pretrained on publicly available Internet data, they possess basic domain knowledge of widely used CAD programming kernels, making it easier for them to debug code independently. Moreover, the procedural nature of CAD programming languages ensures designs can be precisely recreated, supporting verification and collaboration. This study adopts CAD programming code as the output representation for LLMs. Among available options, CADQuery was selected due to its more extensive online resources and its application programming interface's (API's) user-friendly nature, making it an ideal choice for CAD model representation and LLM pretraining and fine-tuning.

Moreover, when working with complex CAD models, CADQuery scripts can be structured and compressed meaningfully by adopting modular and hierarchical approaches. Modularization involves breaking the script into reusable functions that encapsulate repetitive or parameterized operations, such as creating patterns, arrays, or specific geometric features, allowing for code reuse and improving clarity. Hierarchical organization can further enhance readability by defining high-level assemblies composed of smaller, self-contained subcomponents, ensuring that each section of the script focuses on a specific part of the design. Additionally, leveraging loops and conditional statements can streamline the creation of repetitive or variant geometries, reducing code redundancy. To enhance efficiency, parameterization can be employed to represent variable dimensions, constraints, or configurations, enabling flexible customization without requiring extensive modifications to the script. By combining these strategies, CADQuery scripts can remain concise, maintainable, and scalable, even when representing highly intricate CAD models.

3.3 3D Shape Datasets.

The field of 3D shape recognition has seen significant advances in recent years, driven in part by the increasing availability of 3D CAD data and the development of deep learning techniques. One of the key challenges in this domain is the effective representation and processing of 3D data, which can come in various forms, including point clouds, meshes, voxel grids, and boundary representations (B-rep).

Several popular 3D shape datasets, such as ShapeNet [39], PartNet [40], and ModelNet [41], include a diverse collection of object categories (e.g., chairs, cars, and tables). These datasets have been extensively used in the computer vision and computer graphics communities for geometric learning tasks. However, these datasets have limitations when applied to engineering design, where precise geometric features that carry significant engineering semantics (e.g., sharp edges, exact dihedral angles) are crucial. For example, a chair in ShapeNet might have smooth, rounded edges suitable for visual and aesthetic purposes, but the 3D model lacks the precision required for manufacturing and mechanical analysis. Similarly, some datasets [42–44] that include mechanical components are also limited in the use of real design applications because 3D models are available only as point clouds and voxels.

To facilitate research on deep learning of CAD models, such as B-rep, Willis et al. introduced the first dataset of human-designed CAD geometries paired with their ground truth (GT) CAD programs represented as construction sequences [45]. Another well-known CAD dataset, the ABC dataset [46], incorporates a comprehensive collection of 1 million CAD models. This dataset comprises models with explicitly parametrized curves and surfaces, offering ground truth data for patch segmentation, geometric feature detection, and shape reconstruction. Ramnath et al. [47] presents a holistic framework for automatically generating geometry and performing data validation with finite element analysis. These datasets are beneficial for the geometric learning of CAD models with sharp features and are thus applicable to real design scenarios.

Despite the richness of current 3D shape datasets, none of them is suitable for fine-tuning LLMs in order to generate 3D designs based on CAD programs. There are two main challenges for this: (1) Textual descriptions of the 3D designs must be available and include detailed dimensional information to explicitly represent the design requirements of 3D shape features. (2) There must be corresponding ground truth CAD programs linked to these textual descriptions. In the following sections, we present a new dataset that supports LLM fine-tuning and addresses these two challenges.

4 Dataset

In this section, we first introduce the pipeline developed from the previous study [8] and the initial dataset generated. Then, we conducted an initial experiment to test the zero-shot capability of GPT-4 in generating CAD programs, also known as CAD program codes or scripts. From the initial experiment, we collect the scripts and subsequently develop a pipeline to validate them based on which we create training data for the fine-tuning processes as detailed in Sec. 5. The training data generated for the fine-tuning processes are also incorporated into the initial multimodal dataset from the previous study [8].

4.1 Introduction to the Initial Multimodal Computer-Aided Design Dataset.

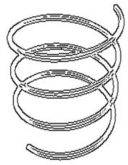
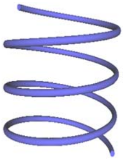
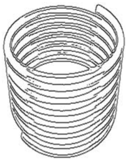
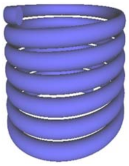
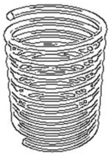
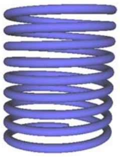
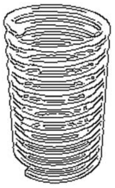
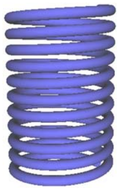
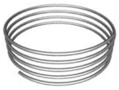
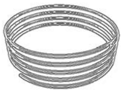
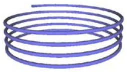
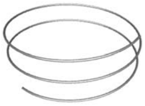
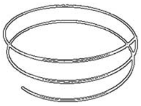
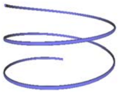
Given the absence of a multimodal dataset that encompasses 1D textual description of part geometries and dimensions, 2D images and sketches, and 3D CAD programs, we have developed a pipeline to generate such data for mechanical components in our previous work [8]. The pipeline automatically synthesizes textual descriptions, images, sketches, and 3D shapes of mechanical components using CAD programming languages (e.g., CADQuery) and computer rendering techniques (e.g., rendering images to sketch-style images using OpenCV). Currently, the dataset contains five common mechanical components: shafts, nuts, flanges, springs, and gears. They are chosen for their popularity in engineering design and their varying levels of complexity, allowing us to test the robustness of LLMs.

The dataset generated from this pipeline comprises 1000 data points for each of the five categories. Table 1 presents an example from the Spring category. Each of the 1000 data points includes a 2D image, a 2D sketch, and a 3D mesh. The 2D image modality is included in the dataset because it is one of the most popular representations often adopted by designers to embody their design ideas and preferences in early design. Moreover, we include the modality of 2D sketches in addition to the 2D images to facilitate the fine-tuning of LMMs.

In addition to images, sketches, and 3D meshes, textual descriptions of the five mechanical components were collected through the crowdsourcing platform, Amazon Mechanical Turk. This approach ensures that our dataset is enriched with various natural language descriptions of the mechanical components, including their dimensional information, which is particularly useful for automatic validation of LLM performance (see Sec. 4.3) and for fine-tuning LLMs (see Sec. 5).

In total, we collected 621 data points for shafts, 671 for nuts, 692 for flanges, 680 for springs, and 661 for gears. These data points with valid textual descriptions are classified as "valid data points." Table 1 displays the 680 valid data points for the springs. It is worth noting that the pipeline described in Ref. [8] can be

Table 1 Examples and definitions of data point

Index	Component name	2D image	2D sketch	3D mesh	Text description
1	Spring_00001				The spring is a helical coil spring with specific dimensional properties. Its coil diameter is 72 mm, and the pitch, which is the distance between adjacent coils, is 6 mm. The free length of the spring, which is the length of the spring when it is not compressed, is 50mm. The wire radius, which indicates the thickness of the wire used to make the spring, is 2.35 mm.
2	Spring_00002				It is a coiled spring wire. The wire has a radius of 2.58 mm, with a coil diameter of 82 mm, and has a free length of 39 mm. There is an 6 mm pitch between the coils of the wires.
3	Spring_00003				The spring is a coiled metal with a 40 mm coil diameter and an 6 mm pitch. Its free length is 53 mm, with a 1.61 mm wire radius.
...	...	...	...	...	...
679	Spring_00679				The spring is a helical coil with a diameter of 32 mm and a pitch of 5 mm. Its free length, when not under compression, is 55 mm. The wire used in the spring has a radius of 1.62 mm.
680	Spring_00680				None
...	...	...	...	...	...
1000	Spring_01000				None

generally applied to obtain data on any other mechanical designs according to user requirements.

4.2 Initial Experiment Setup.

After obtaining the initial dataset earlier, we developed a script generation and evaluation pipeline, illustrated in Fig. 1(*a*). The experiment was carried out using the GPT-4 models' API and CADQuery to generate CAD program scripts. To interact with the GPT-4 API model, we assign a persona to it, defining it as an AI assistant specialized in designing 3D objects. This persona also ensures that GPT-4 exclusively uses CadQuery and restricts the output format to facilitate further validation. The given prompt is "Generate CadQuery code to construct the specified mechanical component. The code must exclusively utilize CadQuery and cannot incorporate any other CAD design packages or software, ultimately exporting the component as an STL file." The resulting CAD program is subsequently converted into 3D shapes for analysis. Similar to the generation of 3D shapes for data points in Table 1, we also use version 2.3.1 for CADQuery in this study.

To improve the performance of GPT models, we proposed a debugger [8], as shown in Fig. 1(*b*). The "debugging process" is an iterative procedure dedicated to the identification and correction of errors found in the generated CAD programs. The "forward pass" ends after executing the generated CAD program scripts to obtain 3D CAD models, regardless of whether the execution is successful or not. This is used to test the zero-shot capability of GPT models. In the event of a successful execution, the process will terminate. An unsuccessful execution indicates the presence of syntax errors within the program, which require the activation of the debugger. Syntax errors encompass a spectrum of programming language misuse, from typographical errors to misapplication of language constructs. For the "debugging process," the previous conversation content (including the user requirements and GPT's responses) and the current error messages are all fed to the same API. This recursive process is imperative to refine the CAD program, ensuring its accuracy and reliability when generating 3D CAD models.

4.3 Script Evaluation Metrics.

We use two metrics to quantify the capabilities of GPT-4 models in generating the CAD program scripts and the corresponding 3D CAD models: parsing rate and intersection over union (IoU). The parsing rate evaluates the extent to which the generated CAD program scripts can be parsed successfully without errors. Upon successful parsing, the quality of the resulting 3D shapes is measured against the GT shapes by calculating the resulting IoU score. IoU is a critical measure of geometric accuracy and quantifies the overlap between the generated shape and the GT shape as a ratio of their intersection to their union. This metric is widely used and is particularly insightful in evaluating the geometric fidelity of the generated designs relative to GT [48].

Since our focus is on shape geometry rather than the positions within a given space, we first align the principal axes of the generated shapes with those of the GT shapes by rotation. We then translate the generated shapes to align their centroids with those of the GT shapes. This transformation process, shown in Fig. 2, ensures that the calculation of the IoU is based only on the geometric

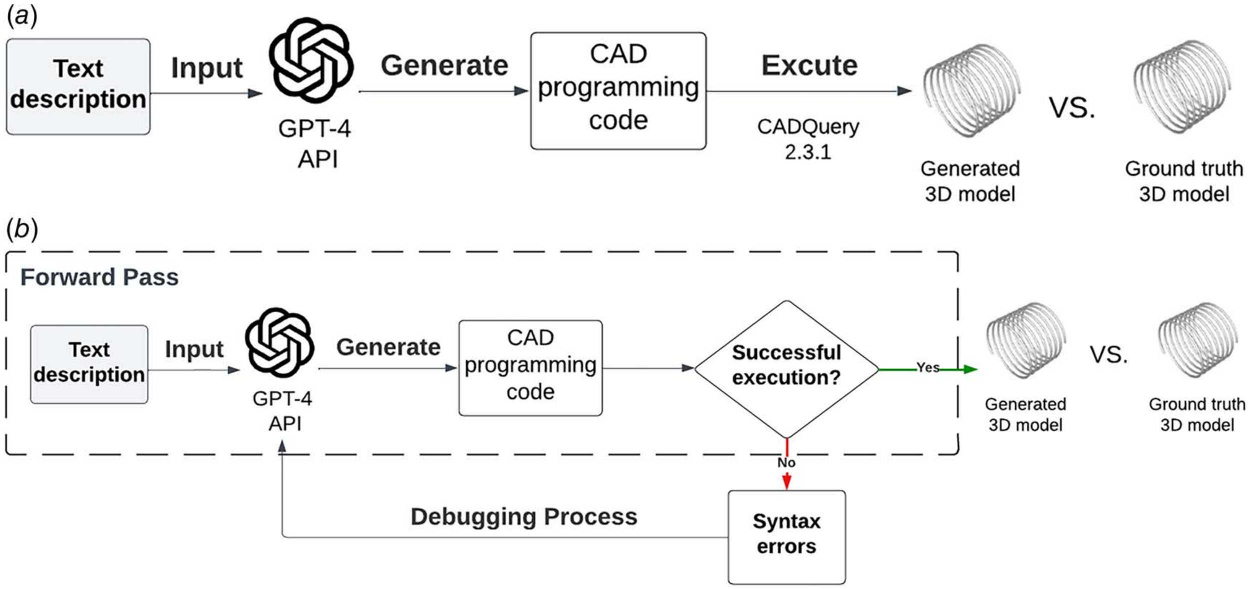


Fig. 1 The pipeline for script generation and evaluation by leveraging the zero-shot capability of the GPT-4 model: (*a*) the pipeline of generating CAD program with GPT-4 model and (*b*) the pipeline of generating CAD program with debugging process

accuracy of the shapes, excluding any discrepancies that might arise from their positioning or orientation.

4.4 Generation of Training Dataset for Large Language Model Fine-Tuning. After the initial experiment, we collected a series of CAD program scripts generated by GPT-4, as shown in Fig. 3, step 1. In support of the development of fine-tuned LLMs for this specific 3D CAD generation task, we prepare a training dataset that includes natural language descriptions of those mechanical components along with their corresponding valid CAD programs. A valid CAD program means that the program can be successfully parsed to generate a 3D CAD model (in mesh representation). The objective of the fine-tuning process is to enhance the GPT model's ability in generating more successful CAD program scripts, which initially GPT-4 struggled to achieve. It is important to note that each valid data point of a mechanical component has a unique text description sourced from Amazon Mechanical Turk. The fine-tuning process uses those valid data points yet unable to correctly generate 3D CAD programs in the initial experiment. These data points are referred to as "failed data points," which are then used to obtain the training dataset based on the pipeline shown in Fig. 3. The objective of Fig. 3 is to pair a valid GPT response with failed data points to compose the training dataset for the fine-tuning purpose. Fine-tuning the GPT model with these failed data points could enhance its ability to generate valid CAD programs that it initially struggled with.

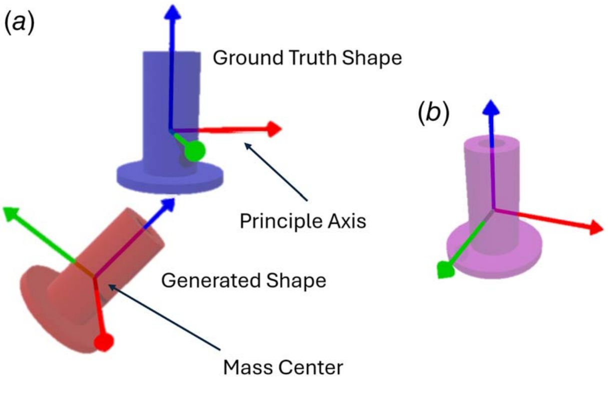
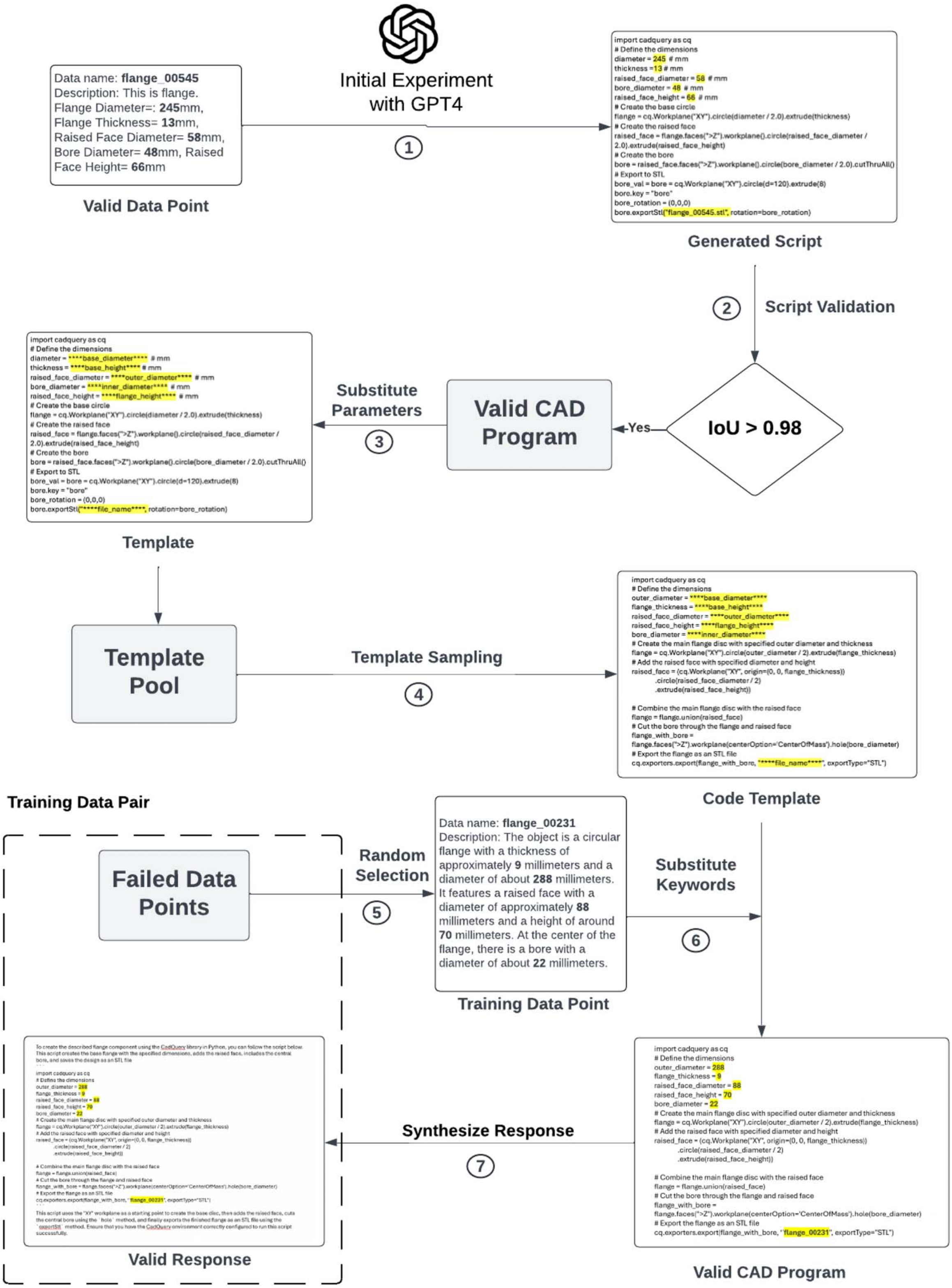


Fig. 2 IoU calibration method: (*a*) before calibration and (*b*) after calibration

In this study, a CAD program is valid only if the generated 3D CAD model has more than 98% overlaps with its GT. With such a threshold, we collect all the valid CAD programs generated by the initial experiment. In order to correct those failed data points and make them valid, we developed a *code template-based* approach. A code template includes string variables, which can be substituted with the actual parameters of each data point (see Fig. 3). We used valid CAD programs generated by the GPT models in the initial experiment to derive code templates. First, we detect the number of parameters in the script of a valid CAD program and then replace these parameters with string variables. This process is illustrated in Step 3 in Fig. 3. As a result, a collection of code templates was assembled, also referred to as a template pool, for each category of the five mechanical components. The quantity of these templates in each pool is presented in Table 2. In particular, the shaft category encompasses four different types of shafts, each with two to five sections, as shown in Fig. 4.

After creating the code template pool, 60 templates are sampled from the corresponding template pool for each category as depicted in step 4 of Fig. 3. Additionally, 60 training data points are randomly selected from the failed data points for each category to ensure that all training data points are indeed failed data points. We sample a small subset of training data points rather than using all the valid data points for two key reasons: first, to ensure the availability of unseen testing data, which is essential for evaluating the performance of the fine-tuned models; and second, to assess the efficiency of the fine-tuning process when working with a limited training dataset. Once this selection process is completed, each training data point is paired with a template to further generate the data pairs for LLM fine-tuning. The string variables are then substituted back with the actual parameters, resulting in a valid CAD program for each training data point, as shown in step 6 of Fig. 3.

The response of the GPT model, i.e., the generated CAD program, includes introductory and concluding parts at the beginning and end of a script. To synthesize a response for each training data point, these introductory and concluding parts are added to the valid CAD program to formulate a valid response, as shown in step



Data name: **flange_00545**
Description: This is flange.
Flange Diameter=: **245**mm,
Flange Thickness= **13**mm,
Raised Face Diameter= **58**mm,
Bore Diameter= **48**mm, Raised
Face Height= **66**mm

```
import cadquery as cq
# Define the dimensions
diameter = 245 # mm
thickness =13 # mm
raised_face_diameter = 58 # mm
bore_diameter = 48 # mm
raised_face_height = 66 # mm
# Create the base circle
flange = cq.Workplane("XY").circle(diameter / 2.0).extrude(thickness)
# Create the raised face
raised_face = flange.faces(">Z").workplane().circle(raised_face_diameter /
2.0).extrude(raised_face_height)
# Create the bore
bore = raised_face.faces(">Z").workplane().circle(bore_diameter / 2.0).cutThruAll()
# Export to STL
bore_val = bore = cq.Workplane("XY").circle(d=120).extrude(8)
bore.key = "bore"
bore_rotation = (0,0,0)
bore.exportStl("flange_00545.stl", rotation=bore_rotation)
```

```
import cadquery as cq
# Define the dimensions
diameter = ****base_diameter**** # mm
thickness = ****base_height**** # mm
raised_face_diameter = ****outer_diameter**** # mm
bore_diameter = ****inner_diameter**** # mm
raised_face_height = ****flange_height**** # mm
# Create the base circle
flange = cq.Workplane("XY").circle(diameter / 2.0).extrude(thickness)
# Create the raised face
raised_face = flange.faces(">Z").workplane().circle(raised_face_diameter /
2.0).extrude(raised_face_height)
# Create the bore
bore = raised_face.faces(">Z").workplane().circle(bore_diameter / 2.0).cutThruAll()
# Export to STL
bore_val = bore = cq.Workplane("XY").circle(d=120).extrude(8)
bore.key = "bore"
bore_rotation = (0,0,0)
bore.exportStl("****file_name****, rotation=bore_rotation)
```

```
import cadquery as cq
# Define the dimensions
outer_diameter = ****base_diameter****
flange_thickness = ****base_height****
raised_face_diameter = ****outer_diameter****
raised_face_height = ****flange_height****
bore_diameter = ****inner_diameter****
# Create the main flange disc with specified outer diameter and thickness
flange = cq.Workplane("XY").circle(outer_diameter / 2).extrude(flange_thickness)
# Add the raised face with specified diameter and height
raised_face = (cq.Workplane("XY", origin=(0, 0, flange_thickness))
          .circle(raised_face_diameter / 2)
          .extrude(raised_face_height))

# Combine the main flange disc with the raised face
flange = flange.union(raised_face)
# Cut the bore through the flange and raised face
flange_with_bore =
flange.faces(">Z").workplane(centerOption='CenterOfMass').hole(bore_diameter)
# Export the flange as an STL file
cq.exporters.export(flange_with_bore, "****file_name****", exportType="STL")
```

Data name: **flange_00231**
Description: The object is a circular flange with a thickness of approximately **9** millimeters and a diameter of about **288** millimeters. It features a raised face with a diameter of approximately **88** millimeters and a height of around **70** millimeters. At the center of the flange, there is a bore with a diameter of about **22** millimeters.

```
import cadquery as cq
# Define the dimensions
outer_diameter = 288
flange_thickness = 9
raised_face_diameter = 88
raised_face_height = 70
bore_diameter = 22
# Create the main flange disc with specified outer diameter and thickness
flange = cq.Workplane("XY").circle(outer_diameter / 2).extrude(flange_thickness)
# Add the raised face with specified diameter and height
raised_face = (cq.Workplane("XY", origin=(0, 0, flange_thickness))
          .circle(raised_face_diameter / 2)
          .extrude(raised_face_height))

# Combine the main flange disc with the raised face
flange = flange.union(raised_face)
# Cut the bore through the flange and raised face
flange_with_bore =
flange.faces(">Z").workplane(centerOption='CenterOfMass').hole(bore_diameter)
# Export the flange as an STL file
cq.exporters.export(flange_with_bore, ''flange_00231", exportType="STL")
```

To create the described flange component using the CadQuery library in Python, you can follow the script below. This script creates the base flange with the specified dimensions, adds the raised face, includes the central bore, and saves the design as an STL file

```
import cadquery as cq
# Define the dimensions
outer_diameter = 288
flange_thickness = 9
raised_face_diameter = 88
raised_face_height = 70
bore_diameter = 22
# Create the main flange disc with specified outer diameter and thickness
flange = cq.Workplane("XY").circle(outer_diameter / 2).extrude(flange_thickness)
# Add the raised face with specified diameter and height
raised_face = (cq.Workplane("XY", origin=(0, 0, flange_thickness))
          .circle(raised_face_diameter / 2)
          .extrude(raised_face_height))

# Combine the main flange disc with the raised face
flange = flange.union(raised_face)
# Cut the bore through the flange and raised face
flange_with_bore = flange.faces(">Z").workplane(centerOption='CenterOfMass').hole(bore_diameter)
# Export the flange as an STL file
cq.exporters.export(flange_with_bore, ''flange_00231", exportType="STL")
```

This script uses the "XY" workplane as a starting point to create the base disc, then adds the raised face, cuts the central bore using the `hole` method, and finally exports the finished flange as an STL file using the `exportStl` method. Ensure that you have the CadQuery environment correctly configured to run this script successfully.

Fig. 3 The pipeline of generating training data for LLM fine-tuning

7 of Fig. 3. Following the complete pipeline depicted in Fig. 3, a fine-tuned training set is established consisting of 60 data points for each category. Each training data point includes a pair consisting of a text description and a valid response, as shown in Table 3. These data points are formatted according to OpenAI's *Chat Completion API* and uploaded via the *Files API* for fine-tuning the GPT model.

This dataset adheres to the FAIR principles (findable, accessible, interoperable, reusable). It is stored in the Texas Data Repository, ensuring long-term accessibility. The repository assigns a globally unique and persistent DOI[3] to the dataset, enhancing its findability.

[3] https://doi.org/10.18738/T8/KV7HON

Table 2 The number of code templates for each category of the mechanical components

Mechanical components	Code template pool size
Shafts (2 sections)	391
Shafts (3 sections)	157
Shafts (4 sections)	278
Shafts (5 sections)	308
Nuts	843
Flanges	1801
Springs	28
Gears	50

All data points are indexed, making them easily searchable. Images and sketches are stored in PNG format, while 3D meshes are in the standard tessellation language (STL) format, both of which are widely recognized standards, facilitating data processing and integration. In addition, the dataset includes a README file that provides detailed instructions on how to use and reuse the dataset in the future [49].

5 Fine-Tuned Large Language Models

In this section, we introduce the experimental settings and the results of the fine-tuning experiments using the training set presented in Sec. 4.

5.1 Experimental Settings. Since different sampling strategies in step 4 of Fig. 3 can impact the training dataset with a bias toward certain features, patterns, or distributions, it is important to quantitatively assess such an impact. Given that code length is a critical factor in industry, affecting maintainability, readability, performance, and cost-effectiveness, this study is motivated to examine the effect of the length distribution of the CAD programs in the training dataset on the LLM fine-tuning performance. In particular, four unique distributions of the training set are created. These different distributions are obtained by implementing four different sampling strategies during the template sampling process (i.e., step 4 in Fig. 3).

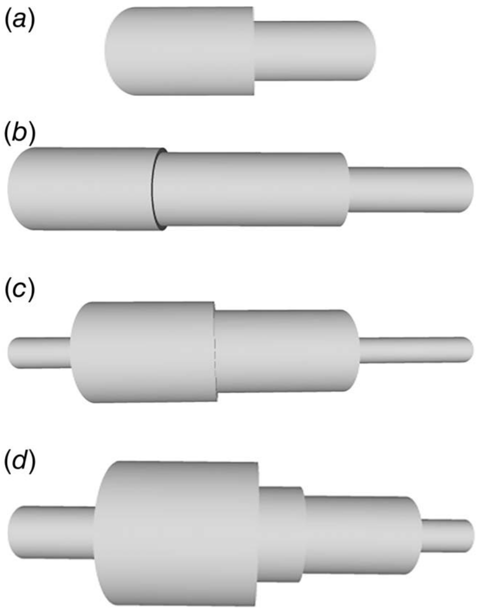


Fig. 4 Example of shaft sections: (*a*) 2 sections shaft, (*b*) 3 sections shaft, (*c*) 4 sections shaft, and (*d*) 5 sections shaft

(a) *Strategy 1*: 60 code templates are randomly chosen from the pool of templates corresponding to each category of the mechanical components.
(b) *Strategy 2*: For robustness of a fine-tuned model, it is essential to include a diverse range of code lengths in the training set. Thus, in each mechanical component category, we divide the templates in each pool into five groups based on their length distribution and the resulting quantiles. Then, in each group, 12 templates are randomly selected.
(c) *Strategy 3*: The 60 templates with the shortest length are chosen from each pool.
(d) *Strategy 4*: The top longest 60 templates are chosen from each pool. This approach explores the impact of using more extensive code structures in the training set on the model performance.

We fine-tuned a GPT-3.5 model using the data sampled with each strategy using the fine-tuning API provided by OpenAI. The fine-tuning process involves three epochs. With a batch size of 1, this requires 900 steps. After 900 training steps, the model was fine-tuned. We designated the models fine-tuned through the data selected using four sampling strategies such as *Fine-tuned Model1*, *Fine-tuned Model2*, *Fine-tuned Model3*, and *Fine-tuned Model4*.

To evaluate the four fine-tuned models, we randomly selected 500 unseen data points (excluding the training data) for each category from the valid data points set as the testing data. Since the shaft has four different categories, we chose 125 data points for each.

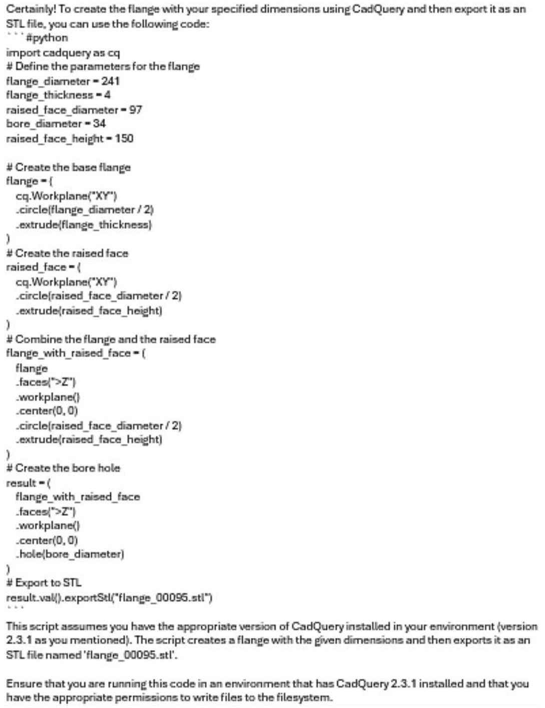
5.2 Results. In this section, we present the results from two different perspectives. First, we compare the GPT-4 model with all fine-tuned models to evaluate the impact of fine-tuning on the parsing rate and IoU scores. In this comparison, we also analyze the effect of fine-tuning in each category of the five mechanical components. Second, we compare the performance of each fine-tuned model to assess the effectiveness of each sampling strategy. Furthermore, we investigate the impact of these strategies in each category of the five mechanical components based on the parsing rate and IoU scores.

5.2.1 Perspective I: Comparison Between GPT-4 and Fine-Tuned Models. We first compare the performance of the GPT-4 model with the fine-tuned models at the aggregated level by computing the average parsing rate of each model regardless of the component category. Figure 5 presents the results as follows: GPT-4 (0.509), Fine-tuned Model1 (0.766), Fine-tuned Model2 (0.779), Fine-tuned Model3 (0.763), and Fine-tuned Model4 (0.671). Overall, every fine-tuned model outperforms the GPT-4 model in terms of parsing rate (indicated by the horizontal dashed line in Fig. 5). The parsing rate reflects the probability of generating syntactically correct scripts. An increase of fine-tuned models' parsing rate compared to GPT-4 suggests that fine-tuning can improve the GPT model's domain knowledge in understanding the CADQuery package and thus increase the likelihood of generating scripts without syntax errors.

Figure 6 shows the parsing rate of each model in each category of the five mechanical components. The results indicate that, on average, the fine-tuned models have a worse performance in generating more complex components (e.g., gears and springs). Moreover, the performance difference between the GPT-4 model and the fine-tuned models is more significant in more complex components than in components with simple geometries, such as the flange, nut, and shaft. This indicates that generating complex components remains more challenging for fine-tuned models than for simple ones. However, it is evident that the fine-tuning processes are more likely to yield more improvement when generating complex geometries than simple ones. It is worth noting that Fine-tuned Model4 performs worse than GPT-4 in the nut category in terms of parsing rate, as shown in Fig. 6 (GPT-4: 0.667 versus Fine-

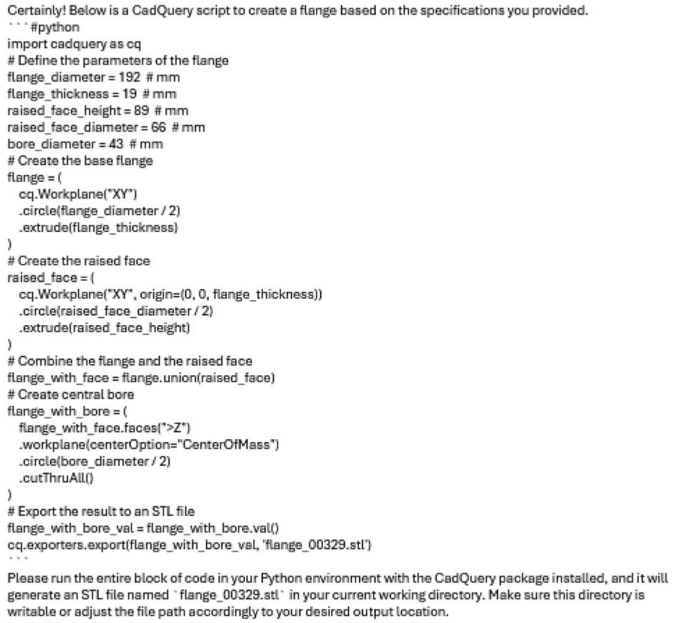
Table 3 Examples of training data pairs

Index	Component name	Text description	Valid response

1	Flange_00095	It is a flange. Flange diameter = 241 mm, flange thickness = 4 mm, raised face diameter = 97 mm, bore diameter = 34 mm, raised face height = 150 mm.	
			Certainly! To create the flange with your specified dimensions using CadQuery and then export it as an STL file, you can use the following code:
			```#python
			import cadquery as cq
			# Define the parameters for the flange
			flange_diameter = 241
			flange_thickness = 4
			raised_face_diameter = 97
			bore_diameter = 34
			raised_face_height = 150

			# Create the base flange
			flange = (
			  cq.Workplane("XY")
			  .circle(flange_diameter / 2)
			  .extrude(flange_thickness)
			)
			# Create the raised face
			raised_face = (
			  cq.Workplane("XY")
			  .circle(raised_face_diameter / 2)
			  .extrude(raised_face_height)
			)
			# Combine the flange and the raised face
			flange_with_raised_face = (
			  flange
			  .faces(">Z")
			  .workplane()
			  .center(0, 0)
			  .circle(raised_face_diameter / 2)
			  .extrude(raised_face_height)
			)
			# Create the bore hole
			result = (
			  flange_with_raised_face
			  .faces(">Z")
			  .workplane()
			  .center(0, 0)
			  .hole(bore_diameter)
			)
			# Export to STL
			result.val().exportStl("flange_00095.stl")
			```
			This script assumes you have the appropriate version of CadQuery installed in your environment (version 2.3.1 as you mentioned). The script creates a flange with the given dimensions and then exports it as an STL file named 'flange_00095.stl'.
			
			Ensure that you are running this code in an environment that has CadQuery 2.3.1 installed and that you have the appropriate permissions to write files to the filesystem.

. . .	. . .	. . .	. . .

60	Flange_00329	This flange is a 89 mm tall, 66 mm wide cylinder with a bore diameter of 43 mm on an 192 mm wide disc that is 19 mm thick.	
			Certainly! Below is a CadQuery script to create a flange based on the specifications you provided.
			```#python
			import cadquery as cq
			# Define the parameters of the flange
			flange_diameter = 192  # mm
			flange_thickness = 19  # mm
			raised_face_height = 89  # mm
			raised_face_diameter = 66  # mm
			bore_diameter = 43  # mm
			# Create the base flange
			flange = (
			  cq.Workplane("XY")
			  .circle(flange_diameter / 2)
			  .extrude(flange_thickness)
			)
			# Create the raised face
			raised_face = (
			  cq.Workplane("XY", origin=(0, 0, flange_thickness))
			  .circle(raised_face_diameter / 2)
			  .extrude(raised_face_height)
			)
			# Combine the flange and the raised face
			flange_with_face = flange.union(raised_face)
			# Create central bore
			flange_with_bore = (
			  flange_with_face.faces(">Z")
			  .workplane(centerOption="CenterOfMass")
			  .circle(bore_diameter / 2)
			  .cutThruAll()
			)
			# Export the result to an STL file
			flange_with_bore_val = flange_with_bore.val()
			cq.exporters.export(flange_with_bore_val, 'flange_00329.stl')
			```
			Please run the entire block of code in your Python environment with the CadQuery package installed, and it will generate an STL file named `flange_00329.stl` in your current working directory. Make sure this directory is writable or adjust the file path accordingly to your desired output location.

tuned Model4: 0.588). This suggests that for simple components like nuts, generating longer CAD programs from a fine-tuned GPT model is more likely to result in worse parsing performance. The reason is that longer CAD programs for simple components often include unnecessary or redundant code, which can introduce errors or complexities that make it harder to parse correctly.

For all CAD program scripts successfully parsed, we calculate the average IoU score of each model, as shown in Fig. 7. The results are as follows: GPT-4 (0.552), Fine-tuned Model1 (0.812), Fine-tuned Model2 (0.824), Fine-tuned Model3 (0.861), and Fine-tuned Model4 (0.799). Overall, all fine-tuned models outperform the GPT-4 model (indicated by the horizontal dashed line in Fig. 7), and this result is statistically significant with all *P*-values <0.01 based on the *t*-test. The higher IoU values again indicate that the fine-tuned models are more capable of understanding natural language descriptions of the mechanical components and potentially better visual and spatial reasoning when processing 2D image and sketch input.

Figure 8 presents the IoU of each model for each category of components. In terms of IoU, each fine-tuned model significantly outperforms the GPT-4 model across all categories, suggesting that fine-tuning can enhance the understanding of human descriptions in CAD generation tasks. For component with simple geometries (e.g., nuts), although Fine-tuned Model4 performs worse than

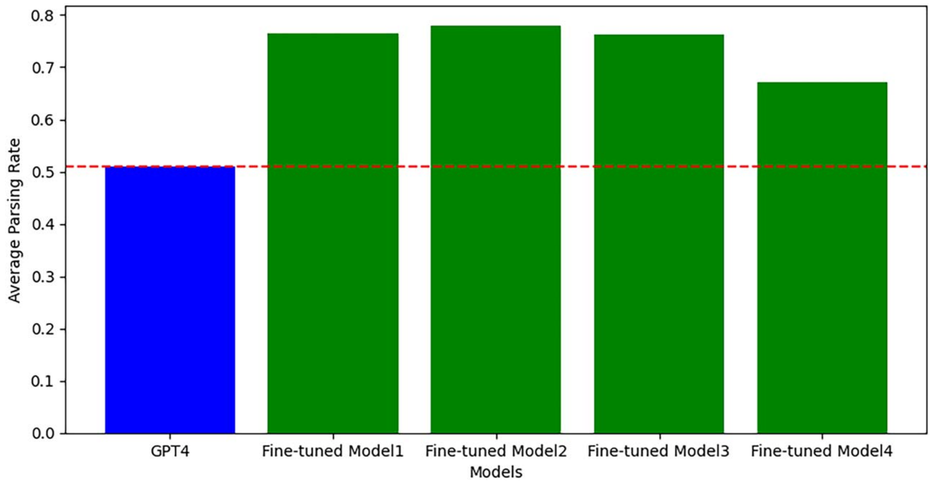


Fig. 5 The results of average parsing rate

GPT-4 in parsing rate, it still outperforms GPT-4 in terms of IoU score (GPT-4: 0.636 versus Fine-tuned Model4: 0.719). This indicates that within the parsed CAD program, Fine-tuned Model4 demonstrates a better understanding of human descriptions compared to GPT-4 and has a stronger capability to translate natural language into a CAD program for the nut component.

For components with complex geometries (e.g., gears), the high IoU values (Fine-tuned Model1: 0.952, Fine-tuned Model2: 0.973, Fine-tuned Model3: 0.981, and Fine-tuned Model4: 0.970) indicate that fine-tuned models have a better understanding of the structure and geometry of gears in general. For springs, although all fine-tuned models show at least a 75-fold improvement in IoU compared to GPT-4, their average IoU scores still remain below 0.5 (IoU score for Spring: GPT-4: 0.0067, Fine-tuned Model1: 0.453, Fine-tuned Model2: 0.467, Fine-tuned Model3: 0.460, and Fine-tuned Model4: 0.492). This suggests that for complex components, fine-tuned models still lack a good understanding of the domain knowledge required to generate accurate CAD programs. To address this, adding more training data pairs would be a potential solution.

5.2.2 Perspective II: Comparison Between Four Fine-Tuned Models. In this section, we compare the four fine-tuned models to investigate the effects of different template sampling strategies on the parsing rate and IoU scores.

As shown in Fig. 5, the Fine-tuned Model2 achieved the highest average parsing rate among all models. The superior performance of Fine-tuned Model2 suggests that fine-tuning a GPT model with an evenly distributed training set can lead to more robust and consistent results across the five mechanical components. This implies that using a diverse yet balanced training set, Fine-tuned Model2 can be better generalized to handle different types of geometries, resulting in higher average parsing rate. This approach also helps mitigate the risk of overfitting, which can occur when a model is trained on a biased dataset. It is also worth noting that Fine-tuned Model4 exhibited the worst performance among all the fine-tuned models. This model was trained on a training dataset sampled using Strategy 4, and longer code templates often contain more complex structures and logic. The poor parsing rate of Fine-tuned Model4 suggests that this added complexity makes it more difficult to accurately learn and generalize patterns, thus increasing the likelihood of syntax errors during the generation of CAD program.

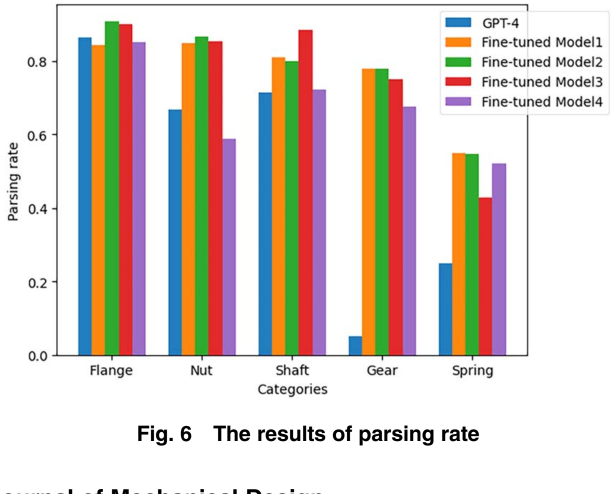


Fig. 6 The results of parsing rate

Figure 6 presents the parsing rates for each model in the five categories of mechanical components. A comparative analysis between Fine-tuned Model1 and Fine-tuned Model2 shows similar parsing rates for shafts (Model1: 0.808 versus Model2: 0.800), gears (Model1: 0.778 versus Model2: 0.778), and springs (Model1: 0.550 versus Model2: 0.546). However, Fine-tuned Model2 exhibited enhanced performance in parsing flanges and nuts compared to Fine-tuned Model1. Additionally, Fine-tuned Model3 consistently outperformed Fine-tuned Model4 in four categories: flanges, nuts, shafts, and gears. This suggests that fine-tuning LLMs with those code templates that have the shortest length (i.e., sampling strategy 3) is more likely to generate syntax error-free codes.

Figure 7 shows that Fine-tuned Model3 (IoU: 0.861) achieves the highest performance compared to the other three fine-tuned models (Model1: 0.812, Model2: 0.824, Model4: 0.799) with the level of statistical significance at 0.05. This indicates that Strategy 3 (using the 60 shortest code templates to fine-tune the LLM) is more effective in generating accurate 3D geometries of the five mechanical components based on the natural language descriptions than the other three strategies. It is also worth noting that Model4 has the lowest IoU scores among the four fine-tuned models, consistent with its performance in the parsing rate. This suggests that with a longer program script, small errors in understanding or generating specific code segments can have a cascading effect, leading to spatial inaccuracies in the overall geometry output. These compound errors contribute to lower IoU scores when comparing the generated geometry with the ground truth. In addition, CAD programs often involve intricate spatial relationships and dependencies between different design elements. Longer code templates increase the number of dependencies that the model needs to understand and replicate, which can be challenging and result in further inaccuracies.

Figure 8 displays the IoU results of each model across the five categories of mechanical components. Among the fine-tuned models, Fine-tuned Model1 and Model2 exhibit comparable

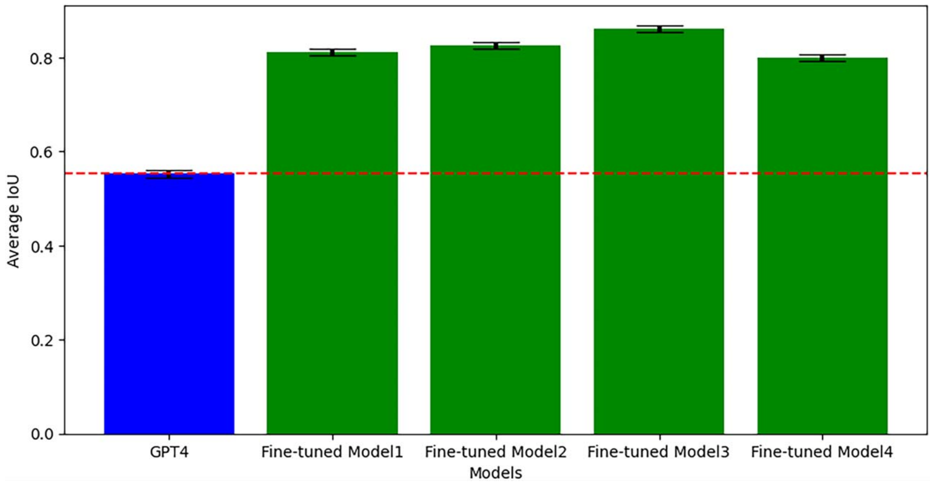


Fig. 7 The results of average IoU

performance across all five categories, as indicated by *t*-tests ($P > 0.05$). This suggests that there are no significant differences in IoU performance between sampling strategy 1 and strategy 2. However, Fine-tuned Model3 significantly outperformed Model4 in the nuts and shafts ($P < 0.05$), verifying the findings of the average IoU performance analysis.

6 Discussion

In this section, based on the results presented, we discuss (1) the effect of fine-tuning and (2) the impact of different sampling strategies when selecting the code templates to train the fine-tune models. At the end of this section, we also share the limitations of the work based on which we propose potential avenues for future research.

6.1 Effect of Fine-Tuning.

Our testing of GPT-4's zero-shot capability revealed limitations in its domain knowledge to generate CAD programs and the associated 3D models of mechanical components, especially for those with higher design complexity, such as gears and springs. The parsing rates observed are only 5.15 % for gears and 25.07 % for springs, as shown in Fig. 6. In contrast, the fine-tuned models achieved at least a 1252 % increase in the parsing rate for the gears and at least a 71.2 % increase in the parsing rate for springs, as shown in Figs. 6 and 8. In terms of IoU, each model demonstrated at least a 75-fold increase in springs and a 58% increase in gears. These results indicate that fine-tuning is an effective method to "teach" LLMs to "learn" the knowledge of CAD programming so as to enhance their performance in generating syntax error-free CAD program scrips and accurate 3D CAD models.

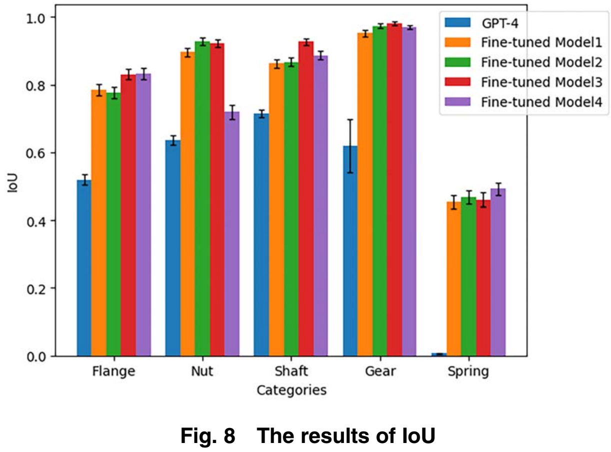


Fig. 8 The results of IoU

To leverage LLMs for specific tasks, fine-tuning and prompt engineering are two common approaches. In prompt engineering, there are primarily two methods: zero-shot learning and few-shot learning. Zero-shot learning is well suited for simple tasks or those that require only general knowledge. However, it is less effective for complex tasks that require domain-specific knowledge or a very specific output format. Few-shot learning, on the other hand, is useful when the model needs to "learn" a new concept or produce a precise data format in output, achieved by providing a few examples in the prompt. However, when generating CAD programs for 3D mechanical components, it is impractical and inefficient for users to provide multiple examples of CAD programs that generate the same component for LLMs to learn. Therefore, fine-tuning is evident to be the most feasible way to improve the performance of GPT in 3D CAD model generation, especially those with complex geometries.

Despite the improvement achieved in the parsing rate and IoU, cost is a factor that must not be neglected in determining the choice of AI services. Based on our experiments and the pricing provided by OpenAI, the cost of generating 1000 components with GPT-4 is approximately $5.55 for input and $22.80 for output (the unit price is $30 per million input tokens and $60 per million output tokens). In contrast, the cost of fine-tuning a single GPT-3.5 model is approximately $4 (using 60 examples for each category of components), and the cost of using the fine-tuned GPT-3.5 for 1000 components is approximately $0.55 for input and $2.28 for output (with the price being $3 per million input tokens and $6 per million output tokens). Therefore, using a fine-tuned model results in a cost saving of approximately ten times. However, it is important to note that this cost-saving analysis does not account for the expenses associated with collecting and preparing the training data required for fine-tuning. Additionally, the conclusion is based solely on GPT-3.5 pricing, which may differ from other models or providers. These limitations highlight the need for a more comprehensive cost analysis to better evaluate the economic feasibility of adopting fine-tuned models in practice.

6.2 Effect of Sampling Strategies

6.2.1 The Effect on Parsing Rate and Intersection Over Union. From the results shown in Secs. 5.2.1 and 5.2.2, Fine-tuned Model2 has the best performance in terms of the average parsing rate. This suggests that by dividing the templates into groups based on their length distribution and selecting an equal number of templates from each group, the fine-tuning dataset effectively covers a broad spectrum of program lengths. This helps the model learn to handle a variety of cases, from short to long CAD programs. Moreover, by covering the entire range of the template pool, strategy 2 ensures that the model is exposed to a comprehensive set of training dataset. This broad coverage allows the model to learn the nuances in CAD programs for various mechanical components, leading to better overall performance.

In terms of parsing rate for each category, it is found that when comparing Fine-tuned Model3 and Model4, Model3 outperforms Model4 in all categories except for spring. This suggests that shorter CAD programs with fewer instructions tend to be simpler and thus less likely to exhibit errors or ambiguities. This simplicity makes it easier for the model to generate output that can be correctly parsed, leading to a higher parsing rate. However, for specific complex components, such as springs, the design may inherently require a longer CAD program to accurately capture detailed features. So, training a model using longer programs allows the model to learn these intricate structures essential to accurately represent such complex geometries.

In terms of average IoU scores, Fine-tuned Model3 outperformed the other fine-tuned models. This suggests that shorter code templates, which focus on the core instructions necessary to generate a geometry, help the model learn the essential steps needed to create accurate 3D geometries. It is also important to note that although Fine-tuned Model3 performs better than Model4 in four categories, it has worse performance in the spring category. One solution to this is to add more spring training pairs to the dataset. Another approach is to recognize that different strategies might be more effective for different components, for example, using strategy 4 to sample the code template for specific complex components such as spring.

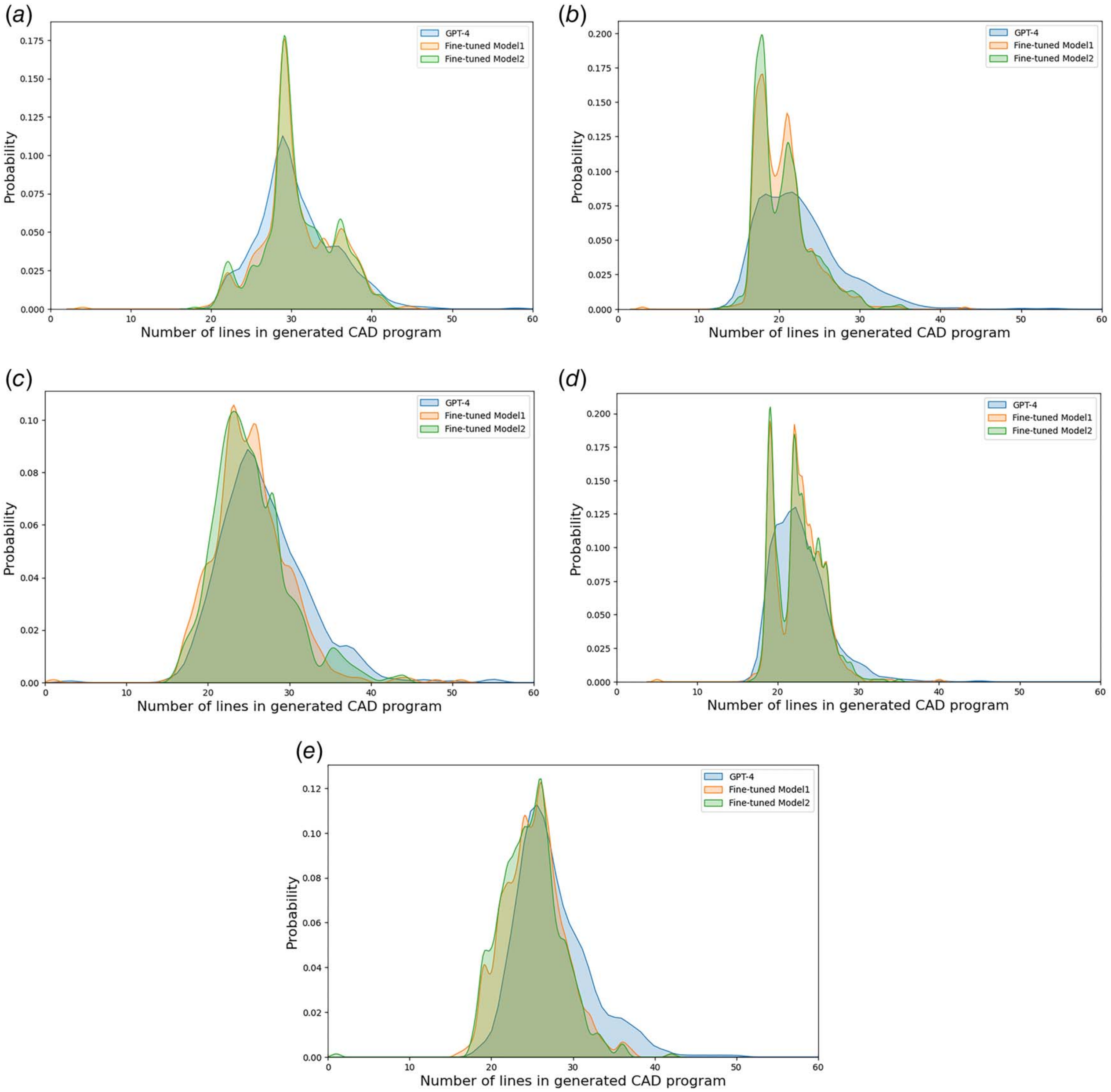


Fig. 9 The probability distribution of number of lines comparison in generated CAD program between GPT-4 and Fine-tuned Model1 and Model2: (*a*) probability distribution of number of lines in CAD program for flange, (*b*) probability distribution of number of lines in CAD program for nut, (c) probability distribution of number of lines in CAD program for shaft, (*d*) probability distribution of number of lines in CAD program for gear, and (*e*) probability distribution of number of lines in CAD program for spring

6.2.2 The Effect on Output Distributions. Figures 9 and 10 illustrate the probability distribution of the length (i.e., the number of lines) of the CAD program codes generated by GPT-4 and the four fine-tuned models. It is important to note that these distribution plots capture all the programs generated by the models regardless of their correctness, i.e., whether they are able to be successfully parsed. In Fig. 9, it is observed that the program length distributions for Fine-tuned Model1 and Model2 are similar and have a high degree of overlap. The overlap area between Fine-tuned Model1 and Fine-tuned Model2 for each category is as follows: flange: 0.977, nut: 0.9613, shaft: 0.959, gear: 0.972, and spring: 0.966. This suggests that despite the differences between sample strategies 1 and 2, the output maintains a consistent distribution pattern. The reason for this is that the original GPT model was pretrained on a vast and diverse code set, which establishes a strong prior distribution over generated CAD programs. This pretrained distribution is quite dominant, and since neither strategy 1 nor strategy 2 introduces a significant deviation from the original GPT-4 distribution, the model continues to reflect the distributional patterns learned during pretraining.

Furthermore, the average length of the programs generated by GPT-4, Fine-tuned Model1, and Fine-tuned Model2 is similar across all categories of components, as shown in Fig. 11. However, the variance in the length of the CAD programs is lower for Fine-tuned Model1 and Model2 compared to that of GPT-4, as indicated in Fig. 12. This indicates that while strategy 1 and strategy 2 can maintain the average length of the output from the GPT-4 model, they fail to preserve the variance. The variance in output reflects the extent of overfitting, and the lower variance observed in Fine-tuned Model1 and Fine-tuned Model2 suggests that these models may be overfitting. The reason is that, although strategy 1 and strategy 2 aim to retain the variance from the original GPT-4 output, the training data pairs still narrow the model's focus to valid CAD programs rather than the broader range of the original outputs. To address this decrease in variance, increasing the number of training data would be a potential solution.

To evaluate the effectiveness of strategy 3 and strategy 4 in fine-tuning, we analyze the probability distribution of the length of the CAD programs generated by each strategy. Figure 10 shows the distribution of the length of the CAD programs generated by GPT-4, Fine-tuned Model3, and Fine-tuned Model4. A significant shift between the distributions can be observed in the categories of flange, nut, and shaft. However, the length distributions in the gears and springs do not

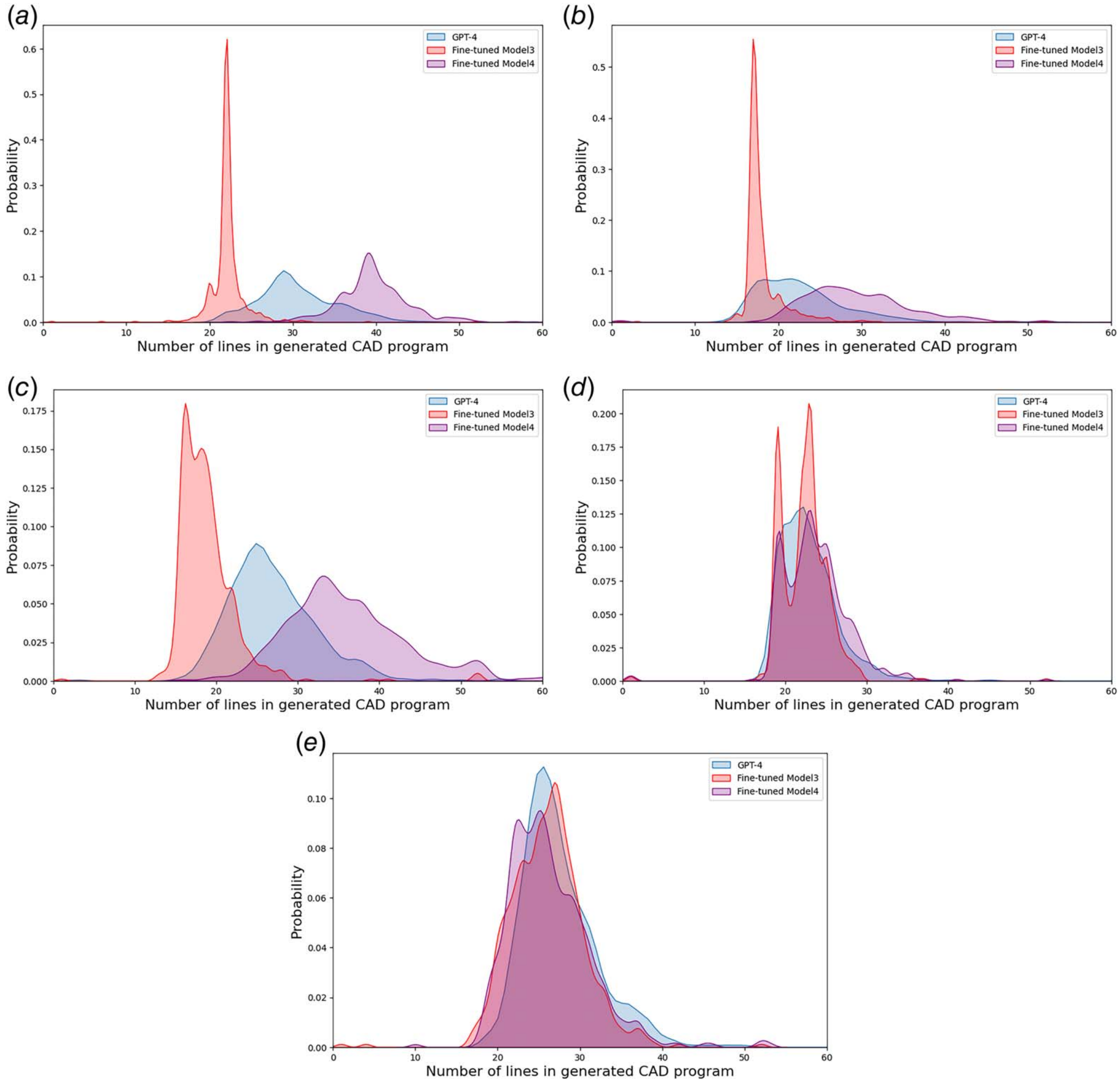


Fig. 10 The probability distribution of number of lines comparison in generated CAD program between GPT-4 and Fine-tuned Model3 and Model4: (a) probability distribution of number of lines in CAD program for flange, (b) probability distribution of number of lines in CAD program for nut, (c) probability distribution of number of lines in CAD program for shaft, (d) probability distribution of number of lines in CAD program for gear, and (e) probability distribution of number of lines in CAD program for spring

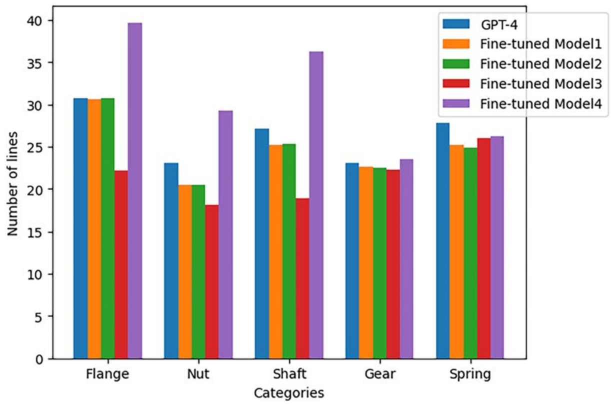


Fig. 11 The average number of lines of codes

exhibit such a shift. The reason for this is twofold. First, the variance of the GPT-4 output in the gear and spring categories is relatively small. Second, the size of the template pools for these two categories (i.e., 50 for gear and 28 for spring), as shown in Table 2, is less than the size of the training set (i.e., 60 programs). Consequently, there are a limited number of code templates for fine-tuning. Thus, the templates selected for gear and spring by strategies 3 and 4 encompass all the templates in the template pool, which is why they do not show significant differences compared to the GPT-4 output.

The shift in the flange, nut, and shaft output distribution indicates that the GPT model is highly sensitive to the fine-tuning data. By training on the shortest or longest CAD programs, the model has learned to prioritize generating shorter or longer outputs, reflecting the characteristics of the fine-tuning dataset. The fine-tuning process of Fine-tuned Model3 and Fine-tuned Model4 successfully adapted the GPT model to produce outputs that align with the specific characteristics of the training data. This also explains why no significant shift is observed in the length distributions of the CAD programs generated by Fine-tuned Model1 and Model2, as sampling strategies 1 and 2 do not significantly change the distribution of the training dataset.

Figure 11 also shows that Fine-tuned Model3 produces the shortest CAD programs in the flange, but, and shaft categories, while Fine-tuned Model4 produces the longest CAD programs in these categories. The average number of lines of codes for gears and springs does not show significant differences between the four fine-tuned models. As mentioned earlier, this is because the template pool size for gears and springs is smaller than the size of the training set. Figure 12 indicates that Fine-tuned Model3 has the smallest variance in the five categories, while Fine-tuned Model4 has the largest variance in these categories. Low variance means that the outputs become similar or consistent because it is overly reliant on the specific examples seen during training. This phenomenon, known as overfitting, means that the model lacks flexibility and adaptability, often producing similar outputs despite slight input variations. Therefore, when selecting a model, our objective is to balance the parsing rate and IoU scores with the output variance to ensure robust results.

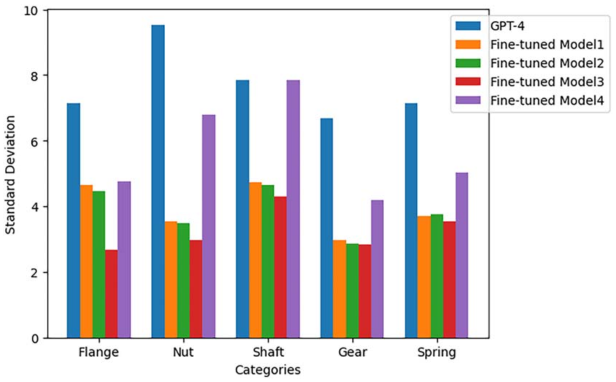


Fig. 12 The standard deviation of the number of lines of codes

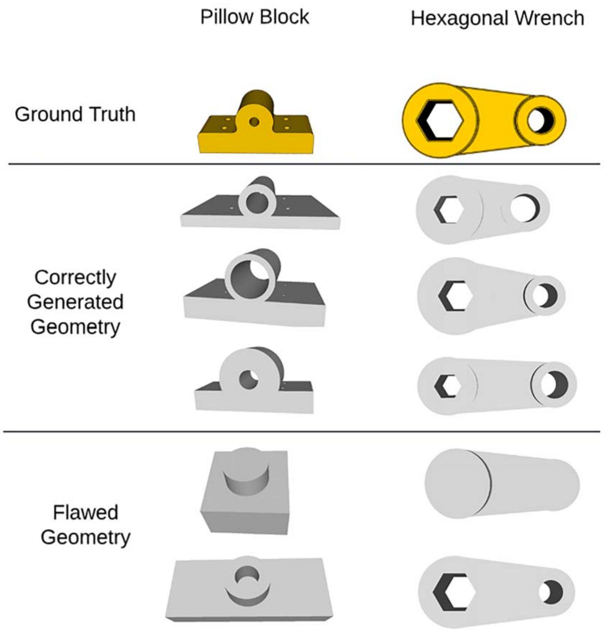


Fig. 13 Pillow block and hexagonal wrench components

From the aforementioned observations, we can conclude that the use of shorter CAD programs to train fine-tuned LLMs can achieve better performance in terms of parsing rate and IoU. Moreover, fine-tuning with shorter program lengths can effectively influence the output length of the fine-tuned model, and since the cost of fine-tuning is based on both the size of the training data and the length of the output, as mentioned in Sec. 6.1, strategy 3 offers a cost-effective solution. However, a trade-off exists because a small variance leads to a higher possibility of overfitting. On the other hand, using longer code to train fine-tuned LLMs (e.g., strategy 4) does not necessarily guarantee better performance compared to other strategies, but the higher variance associated with longer code reduces the likelihood of overfitting.

6.3 Generalizability and Scalability. To demonstrate that the generalizability of our methodology and show it can scale to more complex components, we present two additional real-world components selected from the ABC dataset [46]—pillow block and hexagonal wrench—as shown in the "Ground Truth" row of Fig. 13. The pillow block requires seven design parameters, while the wrench requires 6. We generated 1000 components with varying parameters for each category. We used Llama 3.2 to generate text descriptions from component images and manually filtered these descriptions. In total, we collected 480 descriptions for the pillow block and 572 for the wrench.

With these data, we conducted two experiments: (1) To test the generalizability of our fine-tuned model, we applied Fine-tuned Model2 (trained from the five mechanical components presented in the original manuscript) on the new unseen data (i.e., the pillow block and hexagonal wrench). We then tested the parsing rate and IoU of Fine-tuned Model2, with the results shown as blue bars in Figs. 14 and 15. (2) To test the generalizability of our methodology, we created 60 fine-tuning pairs for both pillow block and hexagonal wrench categories and further fine-tuned Fine-tuned Model2. The improved results are represented in Figs. 14 and 15. Figure 13 shows the qualitative results of flawed geometry generated by fine-tuned Fine-tuned Model2. The significant increase in parsing rate and IoU demonstrates that our methodology has the potential to scale effectively for more complex components, validating its ability to handle diverse CAD models.

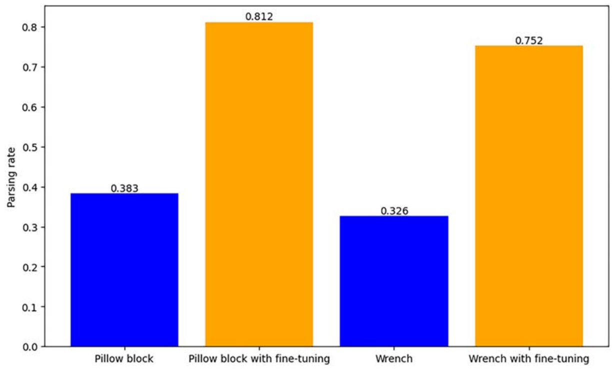


Fig. 14 Parsing rate result

6.4 Limitations and Future Work. In this study, we evaluated the performance of fine-tuned LLMs in generating 3D CAD models of five representative categories of mechanical components with varying geometric complexities. Although the insights gained from the current synthesized dataset are valuable, we acknowledge that the sample size is relatively small compared to the wide array of mechanical components. To obtain an in-depth understanding of the role LLMs play in the generation of 3D CAD models, expanding the CAD dataset is essential. Additionally, further analysis on additional CAD programming languages, such as OpenSCAD and the Fusion 360 PYTHON API, as well as other LLMs like Google's Gemini, will help provide a more comprehensive view of the capabilities of LLMs in the generation of 3D CAD models.

We acknowledge that natural language descriptions have limitations in representing complex geometries, particularly those involving intricate mathematical formulas. So, we do not think the proposed approach is capable of replacing existing standards for full data interoperability between CAD systems. Instead, our approach provides a new angle for solving the problem in fast CAD generation and shows how language-based generative models can assist in this regard for certain geometries, such as standard mechanical components and simple parts with customized features, where procedural scripts can effectively describe a subset of shapes. Rapid generation of such CAD models is expected to save time and reduce costs during the early stages of design.

Furthermore, the potential of fine-tuned MLLMs to generate CAD programs from images or sketches combined with natural language descriptions can also be explored. Our dataset is particularly suitable for generating training data pairs to fine-tune MLLMs, where images combined with natural language descriptions serve as the input request, and CAD programs are generated as the output response.

Given these limitations, our future work will focus on expanding our dataset to include more categories of mechanical components and even assemblies. With a wider range of mechanical components, our dataset will become more practical for industry use and can be applied to fine-tune MLLMs in the future.

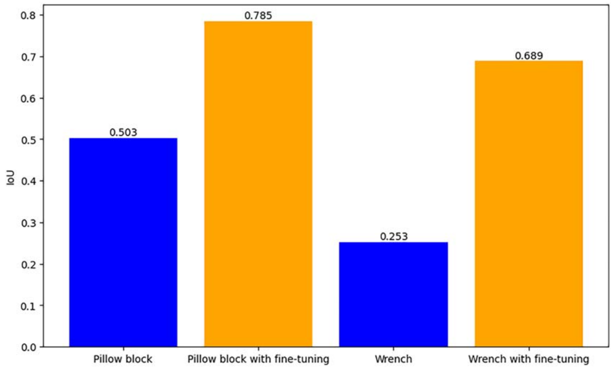


Fig. 15 IoU result

7 Conclusion

In this study, we have introduced a dataset containing multiple design modality of five categories of mechanical components, including shafts, nuts, flanges, springs, and gears, in support of the study of MLLM4CAD. The dataset contains three types of design modality: textual description of part geometries and dimensions, 2D images and sketches, and 3D CAD programs. On the basis of this dataset, we further develop a framework that can synthesize valid GPT responses with failed data points using a novel code template method to generate training data for fine-tuned LLMs. To understand the impact of different strategies in sampling code templates on the fine-tuned model performance, this article investigated four different strategies based on the length distribution of the 3D CAD programs generated by GPT. In particular, we compared the performance of the four fine-tuned models with different sampling strategies with the zero-shot performance of GPT-4 using metrics such as parsing rates and IoU scores.

The results showed that the fine-tuned LLMs significantly outperformed GPT-4 in both the parsing rate and IoU, especially for mechanical components with complex geometries, such as gears and springs. Among the four fine-tuned models, it was found that Fine-tuned Model2 that was trained based on CAD program data created from evenly sampled code templates yielded the best parsing rate. Fine-tuned Model3 that was trained based on CAD program data created from code templates with smallest number of lines exhibited the best performance in IoU. On the basis of these results, we conclude that the fine-tuning process with our dataset can significantly improve the performance of LLMs in text-to-3D CAD generation. Notably, the new knowledge gained from the comparative study on the four different fine-tuned models provided us with valuable guidance on the selection of sampling strategies to build training datasets in fine-tuning practice considering the trade-off between part complexity, model performance, and cost.

The dataset, "Multimodal Dataset for Computer-Aided Design (CAD) Model Generation," is owned by the University of Texas at Austin and was created with contributions from Dr. Zhenghui Sha's research team. The collection of natural language description data was carried out with the approval (STUDY00005512) of the Institutional Review Board of the University of Texas at Austin, ensuring that ethical standards were met, including obtaining informed consent from all participants. Permission for data sharing was secured, and the dataset is available for unrestricted use under a CC0 1.0 Universal (Public Domain Dedication) license through the Texas Data Repository.[4]

Funding Data

- The National Science Foundation (Award 2207408).

Conflict of Interest

There are no conflicts of interest.

Data Availability Statement

The data and information that support the findings of this article are freely available.[5]

[4] See Note 3.
[5] See Note 3.

References

[1] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., et al., 2020, “Language Models are Few-Shot Learners,” NIPS’20: Proceedings of the 34th International Conference on Neural Information Processing Systems, Article No. 159, pp. 1877–1901.

[2] Kasneci, E., Sessler, K., Küchemann, S., Bannert, M., Dementieva, D., Fischer, F., Gasser, U., et al., 2023, “ChatGPT for Good? On Opportunities and Challenges of Large Language Models for Education,” Learning and Individual Differences, **103**, p. 102274.

[3] Kocaballi, A. B., 2023, “Conversational Ai-Powered Design: Chatgpt as Designer, User, and Product,” arXiv preprint 2302.07406.

[4] Filippi, S., 2023, “Measuring the Impact of Chatgpt on Fostering Concept Generation in Innovative Product Design,” Electronics, **12**(16), p. 3535.

[5] Ma, K., Grandi, D., McComb, C., and Goucher-Lambert, K., 2023, “Conceptual Design Generation Using Large Language Models,” Proceedings of the ASME 2023 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference. Volume 6: 35th International Conference on Design Theory and Methodology (DTM), Boston, MA, Aug. 20–23, ASME, p. V006T06A021.

[6] Herzog, V. D., and Suwelack, S., 2022, “Data-Efficient Machine Learning on 3D Engineering Data,” ASME J. Mech. Des., **144**(2), p. 021709.

[7] Picard, C., Edwards, K. M., Doris, A. C., Man, B., Giannone, G., Alam, M. F., and Ahmed, F., 2023, “From Concept to Manufacturing: Evaluating Vision-Language Models for Engineering Design,” arXiv preprint.

[8] Li, X., Sun, Y., and Sha, Z., 2024, “Llm4cad: Multi-modal Large Language Models for 3d Computer-aided Design Generation,” ASME J. Comput. Inf.. Sci. Eng., **25**(2), p. 021005.

[9] Lin, X., Wang, W., Li, Y., Yang, S., Feng, F., Wei, Y., and Chua, T.-S., 2024, “Data-Efficient Fine-Tuning for LLM-Based Recommendation,” ArXiv 2401.17197.

[10] Li, X., Wang, Y., and Sha, Z., 2023, “Deep Learning Methods of Cross-Modal Tasks for Conceptual Design of Product Shapes: A Review,” ASME J. Mech. Des., **145**(4), p. 041401.

[11] Song, B., Miller, S., and Ahmed, F., 2023, “Attention-Enhanced Multimodal Learning for Conceptual Design Evaluations,” ASME J. Mech. Des., **145**(4), p. 041410.

[12] Yang, Z., Li, L., Lin, K., Wang, J., Lin, C.-C., Liu, Z., and Wang, L., 2023, “The Dawn of LMMs: Preliminary Explorations With GPT-4V(ision),” arXiv preprint 2309.17421.

[13] Wu, C., Yin, S., Qi, W., Wang, X., Tang, Z., and Duan, N., 2023, “Visual Chatgpt: Talking, Drawing and Editing With Visual Foundation Models,” arXiv preprint 2303.04671.

[14] Yang, Z., Gan, Z., Wang, J., Hu, X., Lu, Y., Liu, Z., and Wang, L., 2022, “An Empirical Study of GOT-3 for Few-Shot Knowledge-Based VQA,” Proceedings of the AAAI Conference on Artificial Intelligence, Chicago, IL and Online, June 27– July 1, 36(3), pp. 3081–3089.

[15] Wang, Z., Li, M., Xu, R., Zhou, L., Lei, J., Lin, X., Wang, S., Yang, Z., Zhu, C., Hoiem, D., Chang, S., Bansal, M., and Ji, H., 2022, “Language Models With Image Descriptors Are Strong Few-Shot Video-Language Learners,” NIPS’22: Proceedings of the 36th International Conference on Neural Information Processing Systems, Article No. 617, pp. 8483–8497.

[16] Shao, Z., Yu, Z., Wang, M., and Yu, J., 2023, “Prompting Large Language Models With Answer Heuristics for Knowledge-Based Visual Question Answering,” Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, Vancouver, BC, Canada, June 20–22, pp. 14974–14983.

[17] Tsimpoukelli, M., Menick, J. L., Cabi, S., Eslami, S., Vinyals, O., and Hill, F., 2021, “Multimodal Few-Shot Learning With Frozen Language Models,” NIPS’21: Proceedings of the 35th International Conference on Neural Information Processing Systems, Article No. 16, pp. 200–212.

[18] Li, J., Li, D., Savarese, S., and Hoi, S., 2023, “Blip-2: Bootstrapping Language-Image Pre-training With Frozen Image Encoders and Large Language Models,” 40th International Conference on Machine Learning, Honolulu, HI, July 23–29, PMLR, pp. 19730–19742.

[19] Alayrac, J.-B., Donahue, J., Luc, P., Miech, A., Barr, I., Hasson, Y., Lenc, K., et al., 2022, “Flamingo: a Visual Language Model for Few-Shot Learning,” NIPS’22: Proceedings of the 36th International Conference on Neural Information Processing Systems, Article No. 1723, pp. 23716–23736.

[20] Hurst, A., Lerer, A., Goucher, A., Perelman, A., Ramesh, A., Clark, A., Ostorw, A., Welihinda, A., Hayes, A., Radford, A., et al., 2023, “GPT-4V(ision) System Card,” arXiv preprint 2410.21276.

[21] Xian, Y., Schiele, B., and Akata, Z., 2017, “Zero-Shot Learning—The Good, the Bad and the Ugly,” Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Honolulu, HI, July 21–26, pp. 4582–4591.

[22] Pourpanah, F., Abdar, M., Luo, Y., Zhou, X., Wang, R., Lim, C. P., Wang, X.-Z., and Wu, Q. J., 2022, “A Review of Generalized Zero-Shot Learning Methods,” IEEE. Trans. Pattern. Anal. Mach. Intell., **45**(4), pp. 4051–4070.

[23] Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I., 2018, “Improving Language Understanding by Generative Pre-Training,” Preprint. https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

[24] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K., 2019, “Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding,” Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT), **1** (Long and Short Papers), pp. 4171–4186.

[25] Howard, J., and Ruder, S., 2018, “Universal Language Model Fine-Tuning for Text Classification,” arXiv preprint.

[26] Makatura, L., Foshey, M., Wang, B., Hähnlein, F., Ma, P., Deng, B., and Tjandrasuwita, M., 2024, “How Can Large Language Models Help Humans in Design And Manufacturing? Part 2: Synthesizing an End-to-End LLM-Enabled Design and Manufacturing Workflow,” Harvard Data Sci. Rev., **5**.

[27] Jiang, S., and Luo, J., 2024, “AutoTRIZ: Artificial Ideation With TRIZ and Large Language Models,” Proceedings of the ASME 2024 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference. Volume 3B: 50th Design Automation Conference (DAC) , Washington, DC, Aug. 25–28, ASME, p. V03BT03A055.

[28] Yuan, Z., Lan, H., Zou, Q., and Zhao, J., 2024, “3D-PreMise: Can Large Language Models Generate 3D Shapes With Sharp Features and Parametric Control?” arXiv preprint.

[29] Wu, F., Hsiao, S.-W., and Lu, P., 2024, “An AIGC-Empowered Methodology to Product Color Matching Design,” Displays, **81**, p. 102623.

[30] Ma, K., Grandi, D., McComb, C., and Goucher-Lambert, K., 2023, “Conceptual Design Generation Using Large Language Models,” Proceedings of the ASME 2023 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference. Volume 6: 35th International Conference on Design Theory and Methodology (DTM), Boston, MA, Aug. 20–23, ASME, p. V006T06A021.

[31] D. Nelson, M., 2023, “Utilizing ChatGPT to Assist CAD Design for Microfluidic Devices,” Lab Chip, **23**(17), pp. 3778–3784.

[32] Brisco, R., Hay, L., and Dhami, S., 2023, “Exploring the Role of Text-to-Image AI in Concept Generation,” Proc. Design Soc., **3**(1), pp. 1835–1844.

[33] Nelson, M. D., Goenner, B. L., and Gale, B. K., 2023, “Utilizing Chatgpt to Assist CAD Design for Microfluidic Devices,” Lab Chip, **23**(17), pp. 3778–3784.

[34] Picard, C., Edwards, K. M., Doris, A. C., Man, B., Giannone, G., Alam, M. F., and Ahmed, F., 2023, “From Concept to Manufacturing: Evaluating Vision-Language Models for Engineering Design,” arXiv preprint 2311.12668.

[35] Jones, R. K., Barton, T., Xu, X., Wang, K., Jiang, E., Guerrero, P., Mitra, N., and Ritchie, D., 2020, “Shapeassembly: Learning to Generate Programs for 3d Shape Structure Synthesis,” Association for Computing Machinery, **39**(6), p. 234.

[36] Chang, T. A., Tu, Z., and Bergen, B. K., 2024, “Characterizing Learning Curves During Language Model Pre-training: Learning, Forgetting, and Stability,” Trans. Assoc. Comput. Linguist., **12**, pp. 1346–1362.

[37] Wu, R., Xiao, C., and Zheng, C., 2021, “DeepCAD: A Deep Generative Network for Computer-Aided Design Models,” 2021 IEEE/CVF International Conference on Computer Vision (ICCV), Montreal, QC, Canada, pp. 6752–6762.

[38] Xu, X., Willis, K. D., Lambourne, J. G., Cheng, C.-Y., Jayaraman, P. K., and Furukawa, Y., 2022, “Skexgen: Autoregressive Generation of CAD Construction Sequences With Disentangled Codebooks,” arXiv preprint, pp. 24698–24724.

[39] Chang, A. X., Funkhouser, T., Guibas, L., Hanrahan, P., Huang, Q., Li, Z., and Savarese, S., 2015, “ShapeNet: An Information-Rich 3D Model Repository,” arXiv preprint 1512.03012.

[40] Mo, K., Zhu, S., Chang, A. X., Yi, L., Tripathi, S., Guibas, L. J., and Su, H., 2019, “Partnet: A Large-Scale Benchmark for Fine-Grained and Hierarchical Part-Level 3d Object Understanding,” Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Long Beach, CA, June 15–20, pp. 909–918.

[41] Wu, Z., Song, S., Khosla, A., Yu, F., Zhang, L., Tang, X., and Xiao, J., 2015, “3d Shapenets: A Deep Representation for Volumetric Shapes,” Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Boston, MA, June 8–10, pp. 1912–1920.

[42] Kim, S., Chi, H.-G., Hu, X., Huang, Q., and Ramani, K., 2020, “A Large-Scale Annotated Mechanical Components Benchmark for Classification and Retrieval Tasks With Deep Neural Networks,” Computer Vision – ECCV 2020: 16th European Conference, Part XVIII, pp. 175 –191. . Glasgow, UK, August 23–28.

[43] Manda, B., Dhayarkar, S., Mitheran, S., Viekash, V. K., and Muthuganapathy, R., 2021, “‘CADSketchNet’—An Annotated Sketch Dataset for 3D CAD Model Retrieval With Deep Neural Networks,” Comput. Graphics, **99**(9), pp. 100–113.

[44] Lee, H., Lee, J., Kim, H., and Mun, D., 2022, “Dataset and Method for Deep Learning-Based Reconstruction of 3D CAD Models Containing Machining Features for Mechanical Parts,” J. Comput. Design Eng., **9**(1), pp. 114–127.

[45] Willis, K. D. D., Pu, Y., Luo, J., Chu, H., Du, T., Lambourne, J. G., Solar-Lezama, A., and Matusik, W., 2021, “Fusion 360 Gallery: a Dataset and Environment for Programmatic CAD Construction From Human Design Sequences,” ACM Trans. Graphics, **40**(4), pp. 1–24.

[46] Koch, S., Matveev, A., Jiang, Z., Williams, F., Artemov, A., Burnaev, E., Alexa, M., Zorin, D., and Panozzo, D., 2019, “ABC: A Big CAD Model Dataset for Geometric Deep Learning,” Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Long Beach, CA, June 15–20, IEEE, pp. 9593–9603.

[47] Ramnath, S., Haghighi, P., Ma, J., Shah, J., and Detwiler, D., 2020, “Design Science Meets Data Science: Curating Large Design Datasets for Engineered Artifacts,” Proceedings of the ASME 2020 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference. Volume 9: 40th Computers and Information in Engineering Conference, Virtual, Online, Aug. 17–19, ASME, p. V009T09A043.

[48] Zheng, Z., Wang, P., Liu, W., Li, J., Ye, R., and Ren, D., 2020, “Distance-IoU Loss: Faster and Better Learning for Bounding Box Regression,” Proceedings of the AAAI conference on artificial intelligence, New York, Feb. 7–12.

[49] Sun, Y., Li, X., and Sha, Z., 2024, “Multimodal Dataset for Computer-Aided Design (CAD) Model Generation,” Texas Data Repository.