www.cad-journal.net
Computer-Aided Design AND APPLICATIONS

Iterative Generation of Feature-based CAD Models Using an LLM with Domain-specific Constraints

Ammon I. Hepworth[1] and John M. Gauch[2]

[1]Southern Virginia University, ammon.hepworth@gmail.com
[2]University of Arkansas, jgauch@uark.edu

Corresponding author: Ammon I. Hepworth, ammon.hepworth@gmail.com

Abstract. This paper presents a method to generate feature-based parametric mechanical CAD models from text input. The method uses domain-specific constraints (DSCs) to guide a large language model (LLM) in the generation of scripts to build models from the CAD API. It also presents a framework for an iterative text-based system that produces parametric CAD models. The method was implemented by developing DSCs for FreeCAD and OpenAI GPT-5.2 and incorporated into a web-based software application, which generates models that can be visualized with an interactive viewer after each iteration. Tests were run comparing various models with and without this method. Results show that a general LLM (like GPT-5.2) alone struggles to build all but the simplest CAD models. However, using DSCs is an effective way to enable a general LLM to generate more complex, feature-based CAD models from text input. An iterative modeling approach is shown to further increase model generation reliability.

Keywords: AI CAD, LLM CAD, text to parametric CAD
DOI: https://doi.org/10.14733/cadaps.2027.19-35

1 INTRODUCTION

Creating detailed 3D computer-aided design (CAD) models from text-based descriptions is an exciting notion. It not only has potential for much faster model creation, but it also democratizes CAD model building, allowing less experienced builders to bring their ideas to life without having to go through the tedious steps of creating a model from the ground up in a complex CAD system.

A substantial amount of research has explored the generation of 3D geometry from natural language using deep learning. However, most of these approaches use voxels, point clouds, and meshes, which are not parametric CAD models and are not suitable to be used for manufacturing [6, 9, 14, 18]. More recent approaches generate boundary representation (B-Rep) to directly create manufacturable solid models, including SolidGen [2] and BrepGen [16]. However, these approaches do not generate editable, feature-based parametric models and are thus less useful for engineering and manufacturing companies that require iterative design.

Recent research has shown that large language models (LLMs) can generate models for feature-based parametric mechanical CAD systems by producing a script using the application

programming interface (API) to build the CAD model programmatically [3, 4, 7, 8, 12, 13]. While promising, this approach has so far only been shown to produce simple CAD models with basic features. Although general LLMs (such as ChatGPT) can produce working code in many domains, generating scripts with a CAD API to produce parametric CAD models is more difficult. CAD models generated by script need to be written in a particular way to build valid geometry, and even scripts that are syntactically correct may cause execution errors.

In addition, the research to date has focused on building a complete model from a single text prompt, as opposed to an iterative approach that uses multiple text prompts to improve and refine models with a feedback loop. An iterative approach is more challenging than previous approaches due to the need for the user to inspect intermediate results and give further instruction after each iteration. The LLM needs to be aware of previous prompts and resulting code to make changes based on new prompts.

This paper presents two complementary and distinct contributions: 1) a method of using rules, or domain-specific constraints (DSCs), to guide a general LLM on how to generate scripts to build models for feature-based parametric mechanical CAD systems; and 2) a framework for an iterative text-based system that produces parametric CAD models, along with an architecture that implements it using a browser-based user interface.

2 BACKGROUND

Seff et al. performed generative modeling of parametric CAD sketches using deep learning trained on a dataset of 15 million sketches [11]. However, Wu et al. were among the first to develop a deep neural network capable of generatively outputting a sequence of CAD operations to directly build native solid CAD models, which they called DeepCAD. They introduced a unified representation for different CAD operations to enable autoencoding with a Transformer network. They released a large solid modeling dataset for training and future research. It is important to note that their approach was limited to only two types of operations: sketches and extrusions [15].

Since then, a few other researchers have published works which also generate CAD models from a sequence of CAD commands, most of which are trained on the DeepCAD dataset. Khan et al. introduce Text2CAD, which claims to be the first generative AI system for taking natural language to directly generate parametric CAD models, which is based on the DeepCAD dataset [5]. Wang et al. introduce CAD-GPT, also based on the Deep CAD dataset, which uses a Multimodal Large Language Model (MLLM), enhanced with special reasoning, that takes text or an image as input to generate a CAD model [13]. While both papers demonstrate the superiority of their approach over DeepCAD, Text2CAD and CAD-GPT are still subject to some of the same limitations of DeepCAD, one of which is only able to use sketches and extrudes.

Other researchers have shown how existing general large language models (LLMs) like OpenAI's GPT-4 can generate 3D geometry. Makatura et al. show how GPT-4, without any finetuning, can generate limited 3D geometry using primitives, OpenJSCAD, and OnShape (with sketch and extrude). However, 3D models generated this way have several deficiencies. Examples include a table with legs floating in space, a bookshelf with the side and shelves in random locations, and a blocky car with the wheels in the wrong orientation [8]. [4] uses the GPT-4 model to generate Rhino3D CAD 3D models from natural language prompts. This is done by having GPT-4 take natural language commands and generate RhinoScript or Grasshopper which can be executed by Rhino3D to directly create the 3D CAD model.

Yuan et al. use GPT-4 to generate Python code to create 3D models in Blender and introduce a dataset called 3D-PreMise to assess it [17]. Li et al. introduce LLM4CAD, which is an approach that uses GPT-4 and GPT-4V to generate CADQuery code to build a 3D CAD model [7]. In a subsequent paper, the same authors fine-tuned the GPT-3.5 model using CAD models of mechanical components to improve the 3D model generation capabilities and compare this fine-tuned model to GPT-4 without fine tuning. Results suggest that fine tuning does improve CAD model generation performance for model types in which the AI model has been fine-tuned [12].

Jones et al. discuss how LLMs struggle to reliably produce procedural geometry in CAD due to the level of spatial and logical reasoning required. The authors introduce a domain specific language (DSL) called AIDL that is a middle layer between the LLM and geometry. AIDL is a symbolic representation of design intent, not geometry itself. In their approach the LLM creates AIDL programs which are compiled by a geometric constraint solver to produce geometry. They show a significant improvement of geometry generation of an LLM when using AIDL compared to less constrained or direct geometry creation [3].

Recently, a few startup companies, including Adamcad, NeoCAD, and Zoo, have emerged that generate parametric, manufacturable models from text prompts, but are in very early stages [1, 10, 19]. These non-academic activities show the emerging relevance of this topic and its potential significance in industry.

3 METHOD

3.1 Domain Specific Constraints

Using a general LLM (e.g. ChatGPT or Gemini) to produce a script that generates a CAD model for a feature-based parametric CAD system (e.g. SolidWorks, OnShape, FreeCAD) from a prompt is simple, but results are highly limited. As shown in the results section, only the most basic scripts can be run without errors, even though the code is syntactically correct. This is because CAD APIs are semantically fragile, requiring very specific feature ordering and valid topological references. The CAD API documentation is often limited, only discussing syntax, but not semantics. Errors often arise during script execution, so they are difficult to anticipate from code alone. Because LLMs are trained on API documentation and script examples, not failure modes, they make mistakes because they lack awareness of how to robustly build a model using the API. Complex models have more features and are thus more likely to fail during execution time.

To overcome some of these challenges a set of rules, called domain-specific constraints (DSCs), are provided to the LLM in the API call, in addition to the user prompt, as additional context to guide a general LLM on how to generate valid and executable scripts. This effectively provides the necessary additional information for the LLM to effectively generate the script, which is missing in the API documentation and code examples that it likely has already been trained on. The DSCs may include constraints related to general modeling, specific features, primitives, sketching, topology references, unions/subtractions, saving/exporting geometry, and model completeness. DSCs can be broadly organized into the following categories:

1. **Task instructions and response format:** Describes the task to perform, how the LLM should format its response, and the completeness requirements.
2. **Environment and tool usage:** Defines which CAD system, APIs, and feature types are permitted, avoiding ambiguous or unreliable parts of the API.
3. **Geometry representation:** Specifies how geometry should be represented in the CAD system, including rules about primitives, sketches, splines, and allowed features.
4. **Feature placement and orientation:** Defines how features should be positioned and oriented to produce valid geometry.
5. **Feature ordering and dependencies:** Specifies the order in which features must be created and how they depend on each other to ensure valid geometry is created.
6. **Geometric constraints and relationships:** Specifies how geometry should be referenced relative to each other and how geometric constraints should be represented.
7. **File output and export:** Defines how the model should be saved and exported so it can be used for visualization and editing.

The categories addressed and the specific rules within each will vary depending on the CAD system, the LLM, and the types of models being generated. Appropriate DSCs are developed

through analysis of the fragile and ambiguous aspects of the specific CAD API and through testing various prompts that generate models of differing complexity. As models fail, more rules are added to enable the DSCs to guide the LLM to reliably create CAD models. Prompting the LLM for a set of constraints based on documentation ambiguity is a good place to start, however the LLM is not likely to produce an effective DSC without testing with models of various descriptions.

3.2 Iterative Text-based CAD Modeling

Design is iterative, and therefore an effective text-based modeling system requires a means to iterate on CAD models during design. By decomposing a complex model description into sequential prompts, each iteration reduces the complexity of the generation task. This allows the user to verify the result at each step, catching both execution errors and unintended geometry before they compound in subsequent iterations. To support this, a user interface is needed to visualize the model after each prompt, including a viewer with the ability to zoom, rotate, and pan the geometry so the user can effectively analyze the model and determine what changes to make next.

In addition to the user interface, an execution engine takes the prompts from the user and builds a more complete prompt to send the LLM using the predefined DSCs, the previous prompts in the design session, the code generated to produce the current model, and the current prompt entered by the user. This aggregated prompt is sent to the LLM via API call and a script to build the model is returned. This code is executed by calling the CAD system in headless mode with the script as input, generating a triangulated model for visualization and a native parametric model the user can download for further editing. These steps are repeated with each additional prompt in the design session to update the interactive view and associated model.

4 IMPLEMENTATION

The methods described above were implemented using FreeCAD as the parametric CAD system, its associated API for scripting, and OpenAI GPT-5.2 as the LLM to generate scripts.

4.1 Domain Specific Constraints

The DSCs for this implementation were developed specifically for GPT-5.2 to generate FreeCAD API scripts to create CAD models. The initial DSCs were created by prompting GPT-5.2 for the constraints it should include, based on where it expected to see failures. Then, the DSCs were tested on various prompts leading to failed script execution in FreeCAD. The DSCs were revised based on how to avoid the specific error that was generated. In some cases, the DSCs became over constrained, requiring a revision. GPT-5.2 was often helpful in making suggestions on what to revise as well. Iterations continued until the DSC was determined "good enough" based on incremental improvements. It is by no means all-inclusive but does represent a significant improvement in the generation of specific types of models as shown in the results section.

The DSCs developed herein are shown below with explanations for each section. They begin with instructions on how the code builds a parametric PartDesign model with an editable feature tree, including clarification that if a rule is violated, the LLM must correct the output before responding as shown below:

```
You are generating FreeCAD 1.0 Python code that builds a parametric PartDesign model
with an editable feature tree. Follow this strictly. If a rule is violated, correct the
output before responding.
```

The specific constraints related to geometry creation are then spelled out in different sections. Note that they are called rules in this context (instead of constraints) to avoid confusion with

geometric constraints. The first set are general environment rules which provide guidelines on how to build a model generally:

```
General environment rules:
    - Use PartDesign and Sketcher only.
    - Create exactly one PartDesign::Body.
    - Create features only inside the PartDesign Body.
    - Do not set Sketch.Support, MapMode, or reference origin planes.
    - Always import: FreeCAD as App, Part, Sketcher, Mesh.
```

Allowed features specify which features are allowed to be used in model creation, avoiding features that are ambiguous in the API:

```
Allowed features:
    - Sketcher::SketchObject
    - PartDesign::Pad
    - PartDesign::Pocket
    - PartDesign::AdditiveBox
    - PartDesign::AdditiveCylinder
    - PartDesign::SubtractiveCylinder
    - PartDesign::SubtractiveBox (if available in the FreeCAD version)
```

Primitive preference rules spell out how FreeCAD should use pads and pockets over primitives:

```
Primitive preference rules:
    - Prefer PartDesign additive primitives over sketch-based Pads whenever the geometry is a simple box or cylinder.
    - Prefer PartDesign subtractive primitives over sketch-based Pockets whenever the cut geometry is a simple cylinder or box.
    - Use sketch-based Pads or Pockets only when the geometry cannot be represented by the available primitives.
```

Sketch rules explain how the LLM should approach sketch geometry and constraints:

```
Sketch rules:
    - Do not use Sketcher constraints.
    - Use sketches only to define non-primitive profiles.
    - Use only line segments and circles.
    - Express geometry explicitly using coordinates and sketch placement.
```

Hole and cut rules clarify how holes should be performed:

```
Hole and cut rules:
```

```
    - If the prompt specifies round holes, use PartDesign::SubtractiveCylinder whenever
possible.
    - If the prompt specifies rectangular cutouts, use PartDesign::SubtractiveBox
whenever possible.
    - Position subtractive primitives using Placement.
    - Set subtractive primitive lengths large enough to fully cut through the solid.
    - For holes in thin features (plates, flanges, ribs), align the subtractive
primitive axis with the feature's thickness dimension so the cut fully penetrates that
thickness.
    - A hole is considered "through" only if it fully penetrates the local feature
thickness.
```

Feature direction rules specify how to control feature directions and feature sequencing rules specify the ordering of features. For example, there is a rule below that states that additive features should be created before subtractive features. This is because PartDesign within FreeCAD requires subtractive features to have a body to subtract from or the feature will fail. If the first feature is a subtractive feature, the model will fail from the start. Although this does prevent the creation of some valid feature sequences (like bosses inside of holes) this rule is in place to prevent more catastrophic failures.

```
Feature direction rules:
    - Control feature direction using feature placement and orientation.
    - Do not use reversed directions, offsets, midplanes, or up-to-face modes.

Feature sequencing rules:
    - Create additive features before subtractive features.
    - After each additive or subtractive feature, set body.Tip to that feature and call
doc.recompute().
```

Export rules ensure robust export of an STL file (to be used for visualization) and a native FreeCAD (FCStd) file (to be used for native CAD file editing, if desired by the user).

```
Export rules:
    - Save an FCStd file.
    - Export an STL using Mesh.export with body or body.Tip only.
    - Do not create Mesh.Mesh objects or call Shape.tessellate.
```

Finally, the output requirements to ensure only code is generated and a completeness rule to specify that the LLM generates the entire model as described.

```
Output requirements:
    - Output code only, no prose or comments.
    - Code must run headless.
    - Save an FCStd file.
    - Export an STL using Mesh.export.
```

```
Completeness rule:
- The model must include all solids and operations explicitly described in the current
prompt.
```

4.2 Iterative Text-based CAD Modeling

Software was written to implement the DSC method and iterative text-based CAD modeling methods into a usable web application that effectively generates FreeCAD models based on user text prompts. The user interface is simple (shown in Figure 1). It includes a text input field with a 3D viewer which shows the updates of the model after each prompt. It allows users to pan, zoom, and rotate the model. The application also includes an output of the code that was generated for the user to interrogate if more details on the construction of the model are desired. A download button is also there to allow the user to download the generated FreeCAD part so they can do further refinements in the native CAD system, as needed. The prompt used to create the model shown in Figure 1 was simply the word "table".

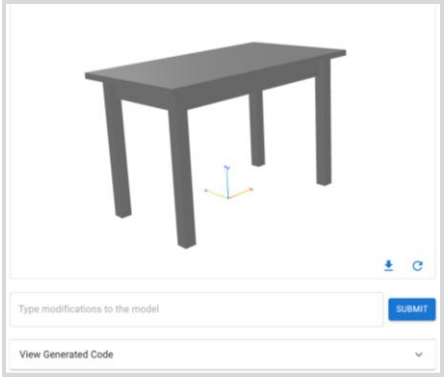

Figure 1: Web UI of the text-based iterative design application.

Iterations to the model are made by entering a description of how to update the model in the field which states: "Type modifications to the model". For example, if a user wanted to modify the table created in Figure 1 to be round, they could enter "make the table round". The model would then update to having a circular top instead of a rectangular one. A user could then enter "make the legs 30% shorter" to make the legs shorter. These last two iterations are illustrated in Figure 2. Iterations to the model can continue to be made until the model is satisfactory to the user.

The architecture of the application consists of the following components shown in Figure 3:

- **Web UI:** user interface for prompt input and model/code visualization
- **App server:** the interface service between the Web UI and the rest of the backend; constructs the API calls using the DSCs
- **Execution service:** runs script in headless FreeCAD and produces STL and FCStd files
- **LLM:** the Open AI API service to execute prompts and produce scripts
- **Model store:** stores the STL and FCStd files

The workflow to generate a visual model from a prompt works as follows (also shown in Figure 3):

1. The user inputs a prompt describing the object they want to build and is sent to the app server on submission.
2. The new prompt is constructed by combining the DSCs with the user prompt, previous prompts, and current model script and is sent to the LLM as an Open AI API call.

3. The LLM sends back the generated script to the app server.
4. The app server sends the script to the execution service to generate the model.
5. Both the STL and FCStd files are sent back to the app server.
6. The app server saves the STL and FCStd files in the model store.
7. The STL and script files are sent to the Web UI for visualization.

When the user clicks the download button, the app server pulls the FCStd file from the model store and sends it to the Web UI for download.

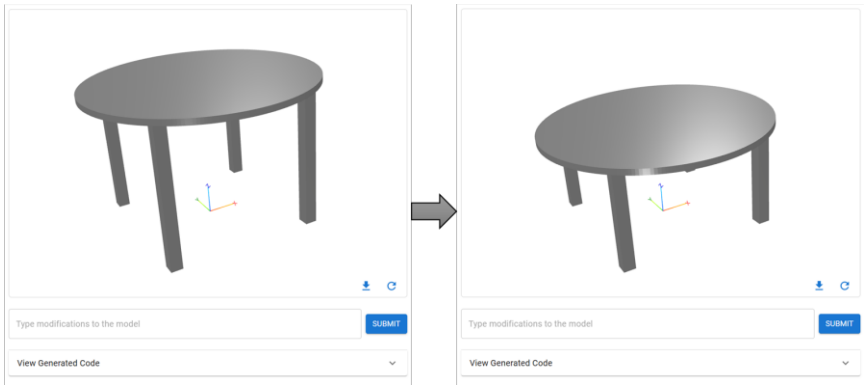

Figure 2: The table model after additional prompts.

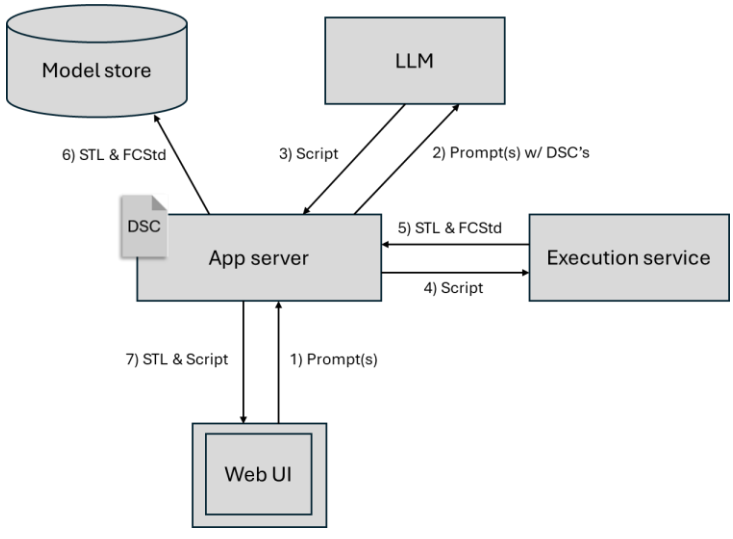

Figure 3: Architecture of the software application.

5 RESULTS

The software described in the implementation was used to test and quantify the improvement in reliability of LLM model generation using DSCs. Two tests were run with a series of different model

descriptions used as inputs. The first test compared the results of models generated with and without the DSCs for primitive geometry described with a single word (e.g. cone, cube, etc.). For the no-DSCs run to give the appropriate outputs, basic additional context (beyond the prompt) was included to the LLM to tell it to generate a FreeCAD 1.0 Python code that builds a parametric PartDesign model with an editable feature tree, to respond with just code, and to produce STL and FCStd files.

The results of the first test are in Table 1 and show that only 50% of the no-DSCs generated models were successfully created, while 83.3% of the DSCs-generated models were created successfully. An additional note is that the failures in the no-DSCs versions produced execution errors, while the single failure in the DSC model created the torus, but did so incorrectly (it was cut in half). The visual results of these generated primitives (with DSCs) are shown in Figure 4, including the incorrectly created torus. This test clearly shows a significant improvement in model generation using DSCs, even in the most basic models.

Test Case	*No DSCs*	*DSCs*
Cone	Pass	Pass
Cube	Fail	Pass
Cylinder	Fail	Pass
Ellipsoid	Pass	Pass
Sphere	Pass	Pass
Torus	Fail	Fail
Percent Passed	50.0%	83.3%

Table 1: DSC vs No-DSC primitive model generation test.

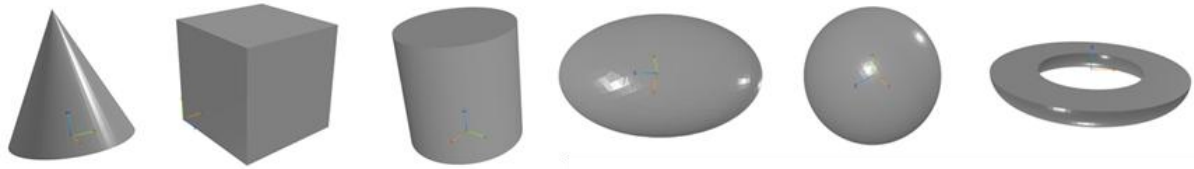

Figure 4: Primitive models generated with DSCs.

The second test consisted of ten models which had prompts that included detailed descriptions of each model (shown in Appendix A). Each description had three distinct sections (separated by semi-colons) that provided additional details. For example, the prompt of the electronics enclosure base is as follows:

> "Open-top rectangular enclosure with outer dimensions 120 × 80 × 30 mm and uniform wall thickness 3 mm; four cylindrical bosses on the interior floor, each 10 mm outer diameter, positioned 10 mm from each interior corner; the bosses have a concentric 4 mm through-hole."

The model generated from this prompt is shown in Figure 5. The image shown is taken directly from the web viewer UI. Figure 6 shows the model opened in FreeCAD, with the feature tree showing the parametric features for the outer box, inner cavity, four holes, and four bosses.

The following code represents the first 10 lines of the code (removing empty lines) generated by GPT-5.2 from the prompt above. For the full code output of the prompt see Appendix B.

```
import FreeCAD as App
```

```
import Part
import Sketcher
import Mesh
doc = App.newDocument("Doc")
body = doc.addObject("PartDesign::Body", "Body")
doc.recompute()
outer = body.newObject("PartDesign::AdditiveBox", "OuterBox")
outer.Length = 120.0
outer.Width = 80.0
...
```

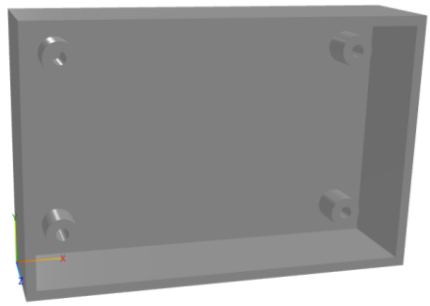

Figure 5: Electronics enclosure base test case in the web viewer UI.

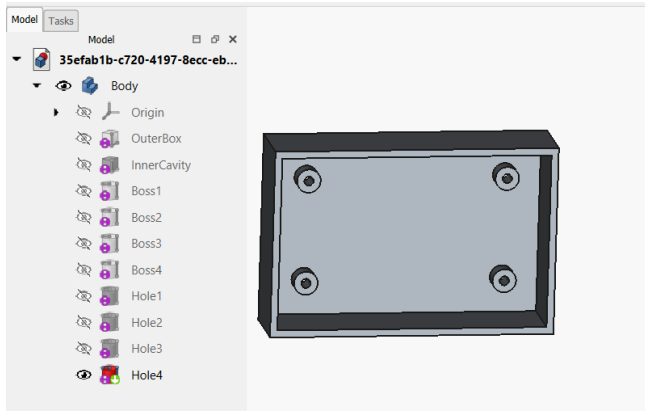

Figure 6: Electronics enclosure base test case in FreeCAD with the feature tree shown.

For both the No-DSCs and DSCs tests, all three portions of the model descriptions were used together in the prompt. A third scenario was tested as well, to see if using an iterative modeling approach (with DSCs) further improved model reliability. To test this, each of the three sections of the description was prompted in order, one at a time in the same design session. If any of the prompts produced invalid results along the way, additional unscripted prompts were given to try to

correct the model. For example, the electronics enclosure base shown above was prompted in three distinct prompts as follows:

1. Open-top rectangular enclosure with outer dimensions 120 × 80 × 30 mm and uniform wall thickness 3 mm
2. four cylindrical bosses on the interior floor, each 10 mm outer diameter, positioned 10 mm from each interior corner
3. the bosses have a concentric 4 mm through-hole

Figure 7 shows the incremental results of each iteration. In this case, the model was produced correctly without additional prompts, but if an invalid result had occurred, additional clarifying prompts could be given to fix the model.

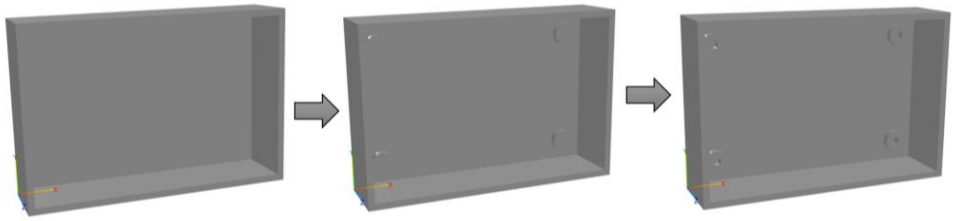

Figure 7: Electronics enclosure base with the iterative modeling approach built in three steps.

Table 2 shows the results of the second test, which generated the ten more complex models. The prompts for each of these models are detailed in Appendix A. Images of the models generated are shown in Figures 8 and 9 for the DSCs and Iterative DSCs tests, respectively, with red circles indicating missing or incorrect features. The percentages shown in the table are defined as the percentage of the model that was created, based on how many of the three sections of the prompt were produced correctly.

For the No-DSCs and DSCs tests, the entire prompt was submitted in a single shot and evaluated on the result of that first run. For the Iterative DSCs tests, each of the three sections was prompted sequentially, with additional corrective prompts allowed up to a maximum of 10 total iterations, at which point the model was evaluated for validity. Note that this means the iterative approach has an inherent advantage in that errors can be corrected mid-process, which is also its primary practical benefit. Also note that each model was not given an exact check of all the dimensions, but rather, a quick visual inspection to verify the features were present. If the features were created in a way that could be interpreted as correct, the feature set was deemed valid.

In the tests shown in Table 2, GPT-5.2 without DSCs did not generate any script that produced a valid model. Meanwhile, both the other methods which used DSCs were able to generate most of the models correctly. The iterative modeling tests indicate that this approach further improves model generation reliability beyond using DSCs alone. Overall, the tests shown in Table 1 and 2 indicate that DSCs not only improve model generation reliability but enable parametric model generation for all but the most basic CAD models.

Test Case	*No DSCs*	*DSCs*	*Iterative DSCs*
Test 1: L-Bracket	0.0%	66.7%	66.7%
Test 2: Stepped Shaft with Keyway	0.0%	66.7%	100.0%
Test 3: Electronics Enclosure Base	0.0%	100.0%	100.0%
Test 4: Mounting Plate with Counterbores	0.0%	100.0%	100.0%

Test 5: Hollow Cylinder with Radial Holes	0.0%	100.0%	100.0%
Test 6: Simple Spur Gear	0.0%	66.7%	100.0%
Test 7: Rectangular Block with Slot and Cross-Holes	0.0%	66.7%	66.7%
Test 8: Cylindrical Hub with Bore and Bolt Circle	0.0%	66.7%	100.0%
Test 9: Rectangular Frame with Cutout and Corner Holes	0.0%	100.0%	100.0%
Test 10: Base Block with Pocket and Vertical Hole	0.0%	100.0%	100.0%
Average	**0.0%**	**83.3%**	**93.3%**

Table 2: Results of the complex model test.

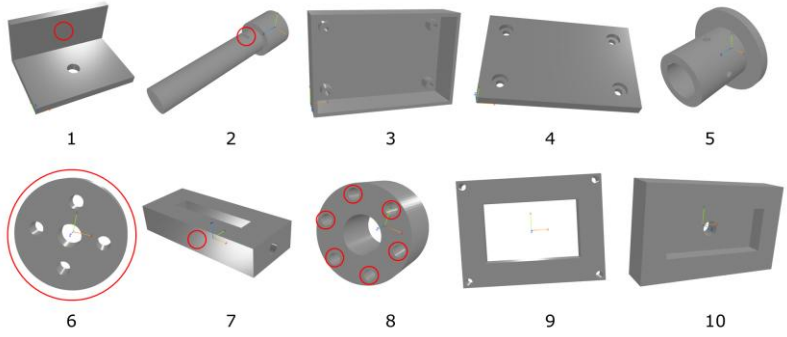

Figure 8: The 10 complex model test cases generated with DSCs.

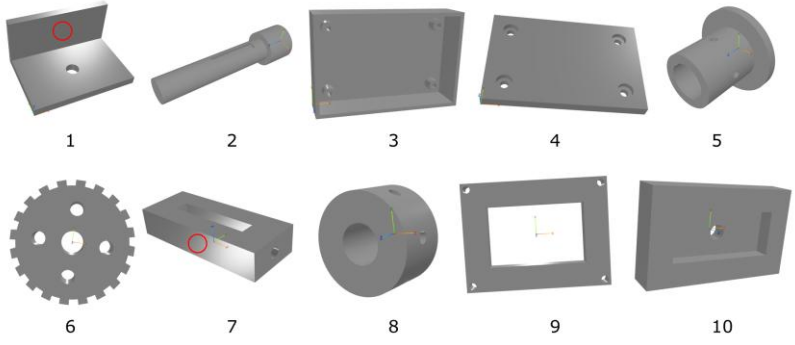

Figure 9: The 10 complex model test cases generated iteratively using DSCs.

6 CONCLUSIONS

This paper demonstrates that domain-specific constraints (DSCs) significantly improve the reliability of LLM-generated scripts for parametric CAD modeling compared to prompting the LLM alone. By encoding modeling semantics as explicit rules, DSCs enable a general LLM to produce executable CAD scripts that build valid and semantically correct geometry. Unlike fine-tuning on CAD-specific datasets, training domain-specific generative models, or developing custom intermediate geometric languages, DSCs require no model training data or geometric solver, making them generally faster and cheaper to implement and adapt to new CAD platforms or LLMs.

The results confirm that DSCs are an effective and practical alternative for generating feature-based parametric CAD models from text prompts, without the overhead of more complex approaches.

The iterative, text-based modeling framework presented here further improves reliability by allowing users to build models incrementally. Rather than attempting to generate a complete model in a single shot, the iterative approach lets users identify and correct errors at each stage through visual inspection, compounding gains in reliability across prompts.

The current implementation is scoped to FreeCAD and GPT-5.2 with a defined feature set. A direct quantitative comparison with other research or commercial text-to-CAD systems was not performed, as differences in supported feature sets, CAD platforms, and evaluation criteria make such comparison methodologically difficult, so this remains an important direction for future work.

Future work should also focus on generalizing DSCs to additional CAD platforms and more complex feature sets, as well as testing the modification of pre-existing models. Combining DSCs with intermediate geometric domain-specific languages is one promising direction for supporting more complex geometric reasoning. Because DSCs require no model training or geometric solver, extending them to new platforms may be more straightforward than other approaches. Automated DSC generation via an LLM feedback loop is a promising approach to streamlining that.

REFERENCES

[1] AdamCAD. https://www.adamcad.com.

[2] Jayaraman, P. K.; Lambourne, J. G.; Desai, N.; Willis, K. D. D.; Sanghi, A.; Morris, N. J. W.: SolidGen: An autoregressive model for direct B-rep synthesis, Transactions on Machine Learning Research, 2023. https://arxiv.org/abs/2203.13944.

[3] Jones, B. T.; Zhang, Z.; Hähnlein, F.; Ahmad, M.; Kim, V.; Schulz, A.: A solver-aided hierarchical language for LLM-driven CAD design, Computer Graphics Forum, Proceedings of Eurographics, 2025. https://doi.org/10.1111/cgf.70250.

[4] Kapsalis, T.: CADgpt: Harnessing natural language processing for 3D modelling to enhance computer-aided design workflows, arXiv preprint arXiv:2401.05476, 2024.

[5] Khan, M. S.; Sinha, S.; Sheikh, T. U.; Stricker, D.; Ali, S. A.; Afzal, M. Z.: Text2CAD: Generating sequential CAD models from beginner-to-expert level text prompts, Advances in Neural Information Processing Systems (NeurIPS), 2024.

[6] Klokov, R.; Boyer, E.; Verbeek, J.: Discrete point flow networks for efficient point cloud generation, arXiv preprint arXiv:2007.10170, 2020. https://doi.org/10.1007/978-3-030-58592-1_41.

[7] Li, X.; Sun, Y.; Sha, Z.: LLM4CAD: Multi-modal large language models for three-dimensional computer-aided design generation, Proceedings of the ASME International Design Engineering Technical Conferences and Computers and Information in Engineering Conference (IDETC/CIE), 2024. https://doi.org/10.1115/DETC2024-143740.

[8] Makatura, L.; Foshey, M.; Wang, B.; Hähnlein, F.; Ma, P.; Deng, B.; Tjandrasuwita, M.; Spielberg, A.; Owens, C. E.; Chen, P. Y.; Zhao, A.; Zhu, A.; Norton, W.; Gu, E.; Jacob, J.; Li, Y.; Schulz, A.; Matusik, W.: How can large language models help humans in design and manufacturing?, Harvard Data Science Review, 2023. https://doi.org/10.21428/e4baedd9.745b62fa.

[9] Nash, C.; Ganin, Y.; Eslami, S. M. A.; Battaglia, P.: PolyGen: An autoregressive generative model of 3D meshes, Proceedings of the International Conference on Machine Learning (ICML), 2020. https://arxiv.org/abs/2002.10880.

[10] NeoCAD. https://www.neocadsrl.com.

[11] Seff, A.; Ovadia, Y.; Zhou, W.; Adams, R. P.: SketchGraphs: A large-scale dataset for modeling relational geometry in computer-aided design, arXiv preprint arXiv:2007.08506, 2020.

[12] Sun, Y.; Li, X.; Sha, Z.: Large language models for computer-aided design fine-tuned: Dataset and experiments, Journal of Mechanical Design, 147(4), 2025. https://doi.org/10.1115/1.4067713.
[13] Wang, S.; Chen, C.; Le, X.; Xu, Q.; Xu, L.; Zhang, Y.; Yang, J.: CAD-GPT: Synthesising CAD construction sequence with spatial reasoning-enhanced multimodal LLMs, arXiv preprint arXiv:2412.19663, 2024. https://doi.org/10.1609/aaai.v39i8.32849.
[14] Wu, J.; Zhang, C.; Xue, T.; Freeman, W. T.; Tenenbaum, J. B.: Learning a probabilistic latent space of object shapes via 3D generative adversarial modeling, Advances in Neural Information Processing Systems (NeurIPS), 2016.
[15] Wu, R.; Xiao, C.; Zheng, C.: DeepCAD: A deep generative network for computer-aided design models, Proceedings of the IEEE International Conference on Computer Vision (ICCV), 2021. https://doi.org/10.1109/ICCV48922.2021.00670.
[16] Xu, X.; Lambourne, J. G.; Jayaraman, P. K.; Wang, Z.; Willis, K. D. D.; Furukawa, Y.: BrepGen: A B-rep generative diffusion model with structured latent geometry, ACM Transactions on Graphics, Proceedings of SIGGRAPH, 2024. https://doi.org/10.1145/3658129.
[17] Yuan, Z.; Lan, H.; Zou, Q.: 3D-PreMise: Can large language models generate 3D shapes with sharp features and parametric control?, arXiv preprint arXiv:2401.06437, 2024.
[18] Zhou, L.; et al.: 3D shape generation and completion through point voxel diffusion, Proceedings of the IEEE International Conference on Computer Vision (ICCV), 2021. https://doi.org/10.1109/ICCV48922.2021.00577.
[19] Zoo.dev. https://zoo.dev.

APPENDIX A: TEST CASE PROMPTS FOR COMPLEX MODEL GENERATION

Test 1: L-Bracket
Right-angle L-shaped solid with a horizontal base 80 × 50 × 6 mm and a vertical flange 80 × 40 × 6 mm joined along the base's long edge; a 10 mm hole in the center of the horizontal base; a 10 mm hole through the center of the vertical flange.

Test 2: Stepped Shaft with Keyway
Cylindrical shaft with total length 120 mm, main diameter 20 mm; a 30 mm diameter shoulder 25 mm long at one end; a rectangular keyway 6 mm wide, 3 mm deep, and 40 mm long in the 20 mm section, starting 10 mm from the shoulder.

Test 3: Electronics Enclosure Base
Open-top rectangular enclosure with outer dimensions 120 × 80 × 30 mm and uniform wall thickness 3 mm; four cylindrical bosses on the interior floor, each 10 mm outer diameter, positioned 10 mm from each interior corner; the bosses have a concentric 4 mm through-hole.

Test 4: Mounting Plate with Counterbores
Rectangular plate measuring 120 × 80 × 8 mm; four through-holes of 6 mm diameter with centers positioned 15 mm from each plate edge; each hole has a concentric counterbore of 12 mm diameter and 4 mm depth.

Test 5: Hollow Cylinder with Radial Holes
Hollow cylindrical tube with outer diameter 60 mm, inner diameter 40 mm, and length 80 mm; four equally spaced radial through-holes of 8 mm diameter located at mid-length around the circumference; a 100 mm flange at one end.

Test 6: Simple Spur Gear
Cylindrical spur gear with 20 equally spaced teeth that are straight and uniform around the circumference with root diameter of 48 mm, and thickness 10 mm; central through-bore diameter 10 mm; four equally spaced circular lightening holes of 6 mm diameter located midway between the bore and the tooth root.

Test 7: Rectangular Block with Slot and Cross-Holes
Solid rectangular block measuring 100 × 40 × 20 mm; a centered rectangular slot on the top face measuring 60 mm long, 10 mm wide, and 10 mm deep; two perpendicular through-holes of 8 mm diameter crossing at the center of the block.

Test 8: Cylindrical Hub with Bore and Bolt Circle
Solid cylinder with diameter 50 mm and length 30 mm; a concentric through-bore of 20 mm diameter; six equally spaced radial through-holes of 6 mm diameter located on a 40 mm bolt circle.

Test 9: Rectangular Frame with Cutout and Corner Holes
Rectangular solid measuring 120 × 90 × 12 mm; a centered rectangular cutout measuring 80 × 50 mm passing fully through the part; four through-holes of 6 mm diameter located at the four outer corners of the frame.

Test 10: Base Block with Pocket and Vertical Hole
Rectangular base block measuring 100 × 60 × 20 mm; a centered rectangular pocket on the top face measuring 70 × 30 mm and 10 mm deep; a vertical through-hole of 10 mm diameter centered within the pocket floor.

APPENDIX B: CODE GENERATED BY LLM FOR ELECTRONICS ENCLOSURE BASE TEST CASE

```
import FreeCAD as App
import Part
import Sketcher
import Mesh

doc = App.newDocument("Doc")

body = doc.addObject("PartDesign::Body", "Body")
doc.recompute()

outer = body.newObject("PartDesign::AdditiveBox", "OuterBox")
outer.Length = 120.0
outer.Width = 80.0
outer.Height = 30.0
outer.Placement = App.Placement(App.Vector(0, 0, 0), App.Rotation(App.Vector(0, 0, 1), 0))
body.Tip = outer
doc.recompute()

inner = body.newObject("PartDesign::SubtractiveBox", "InnerCavity")
inner.Length = 114.0
inner.Width = 74.0
inner.Height = 30.0
inner.Placement = App.Placement(App.Vector(3.0, 3.0, 3.0), App.Rotation(App.Vector(0, 0,
1), 0))
body.Tip = inner
doc.recompute()

boss_positions = [
    (10.0 + 5.0, 10.0 + 5.0),
    (114.0 - 10.0 - 5.0, 10.0 + 5.0),
    (10.0 + 5.0, 74.0 - 10.0 - 5.0),
    (114.0 - 10.0 - 5.0, 74.0 - 10.0 - 5.0),
]

boss_height = 10.0
boss_radius = 5.0

for i, (x, y) in enumerate(boss_positions, start=1):
    boss = body.newObject("PartDesign::AdditiveCylinder", f"Boss{i}")
    boss.Radius = boss_radius
    boss.Height = boss_height
```

```
    boss.Placement = App.Placement(App.Vector(3.0 + x, 3.0 + y, 0.0),
App.Rotation(App.Vector(0, 0, 1), 0))
    body.Tip = boss
    doc.recompute()

hole_radius = 2.0
hole_length = 50.0

for i, (x, y) in enumerate(boss_positions, start=1):
    hole = body.newObject("PartDesign::SubtractiveCylinder", f"Hole{i}")
    hole.Radius = hole_radius
    hole.Height = hole_length
    hole.Placement = App.Placement(App.Vector(3.0 + x, 3.0 + y, -5.0),
App.Rotation(App.Vector(0, 0, 1), 0))
    body.Tip = hole
    doc.recompute()

fcstd_path = "35efab1b-c720-4197-8ecc-ebeed4a99ade.FCStd"
stl_path = "35efab1b-c720-4197-8ecc-ebeed4a99ade.stl"

doc.saveAs(fcstd_path)
Mesh.export([body], stl_path)
```