Text-to-CAD

An AI Co-Pilot for Mechanical Design

Computer-Aided Design (CAD) tools are indispensable in engineering and manufacturing, but they remain **complex and time-consuming** to use. Mechanical engineers must not only understand design principles, but also master the CAD software's scripting or parametric interfaces to create and modify models. As one study notes, "CAD modeling remains a complex, labor-intensive process requiring specialized expertise, particularly in scripting and parametric modeling". This expertise barrier slows design cycles and limits accessibility: writing and debugging precise parametric CAD scripts often demands multiple refinements and significant effort. At the same time, there is a **pressing need for rapid prototyping and generative design** in industry. To meet this demand, AI-driven **Text-to-CAD** systems are emerging: they let users describe 3D shapes or edits in plain English and automatically generate or edit CAD models. In such a co-pilot setup, an embedded assistant converts natural-language commands into CAD API calls or script macros, executing them in the CAD system to update the 3D model.

Integrating Large Language Models (LLMs) like GPT-4 into CAD holds great promise. Recent research proposes frameworks where an LLM generates an initial FreeCAD Python script from a text prompt, then executes and *iteratively refines* it based on errors. Early experiments (e.g. Query2CAD, GAD, LLM4CAD) show that LLMs can handle many simple-to-moderate mechanical designs, significantly reducing the manual scripting workload. Future work even envisions full cloud-based execution and advanced memory mechanisms to streamline Text-to-CAD workflows.

However, developing a robust Text-to-CAD system faces **multiple challenges**. Natural language is inherently ambiguous – interpreting a command like "add a 5 mm fillet on the back edge" requires understanding geometry and context. LLMs are probabilistic and stateless, so they often produce syntactically incorrect or logically flawed scripts on the first try. For example, executing an LLM-generated script may raise *syntax errors*, *invalid boolean operations*, or *wrong API calls*, and the system must detect and handle these gracefully. Ensuring the generated CAD geometry is **valid** (no self-intersections, degenerate features, etc.) is nontrivial. Finally, embedding the AI co-pilot into the CAD UI – whether as a dockable chat panel in FreeCAD or a separate web interface – requires careful design to be intuitive for engineers.

Current Approaches and Research

Text-to-CAD is an active research frontier. Academic prototypes and tools illustrate the state of the art:

- **LLM-driven script generation**. In Query2CAD, an LLM (e.g. GPT-4) translates a user's text into a FreeCAD Python macro. The macro is executed to create a 3D part, and if the result is unsatisfactory, the system uses a vision-and-language model (BLIP2) and even human feedback to refine the prompt and regenerate the script. Similarly, a recent study integrated GPT-4o with an OpenSCAD backend: user queries go to a GPT API, which returns a parametric CAD script; the script is validated and iteratively improved via syntax checking and GPT self-evaluation. These closed-loop LLM frameworks typically use *prompt engineering* and feedback loops to correct errors and hone the design.

- **Text-prompt interfaces**. Some platforms focus on fully automated generation of editable CAD models. For example, the open-source **ZOO Text-to-CAD** tool allows users to enter text prompts and outputs a fully-edited STEP file with boundary representation (B-Rep) geometry. Unlike image-based 3D generators that produce point clouds, ZOO emphasizes *parametric CAD geometry* so designers can further refine the results in existing CAD software.

- **Emerging frameworks**. Other efforts include LLM4CAD (GPT-driven CAD generation with metrics like code parse-rate and IoU accuracy) and AIDL (an AI Design Language that offloads geometry calculations to solvers), showing the trend of combining LLM reasoning with CAD-specific knowledge. Commercial and research tools are in alpha development; for instance, **GAD** (Generative AI-assisted Design) is an open-source Python/Streamlit application where users type requirements and GPT-4o generates an OpenSCAD model script in real-time. These systems often expose users to the intermediate script, enabling manual tweaks.

In summary, current solutions generally fall into **script-generation pipelines**: text ⇒ LLM ⇒ CAD script ⇒ CAD execution ⇒ feedback/verify ⇒ (optionally repeat). This contrasts with direct *geometry generation* approaches (e.g. 3D diffusion models or point-cloud generators), which struggle to produce precise engineering models. By using script output (which can be parameterized and edited), the system ensures results are fully compatible with existing CAD workflows.

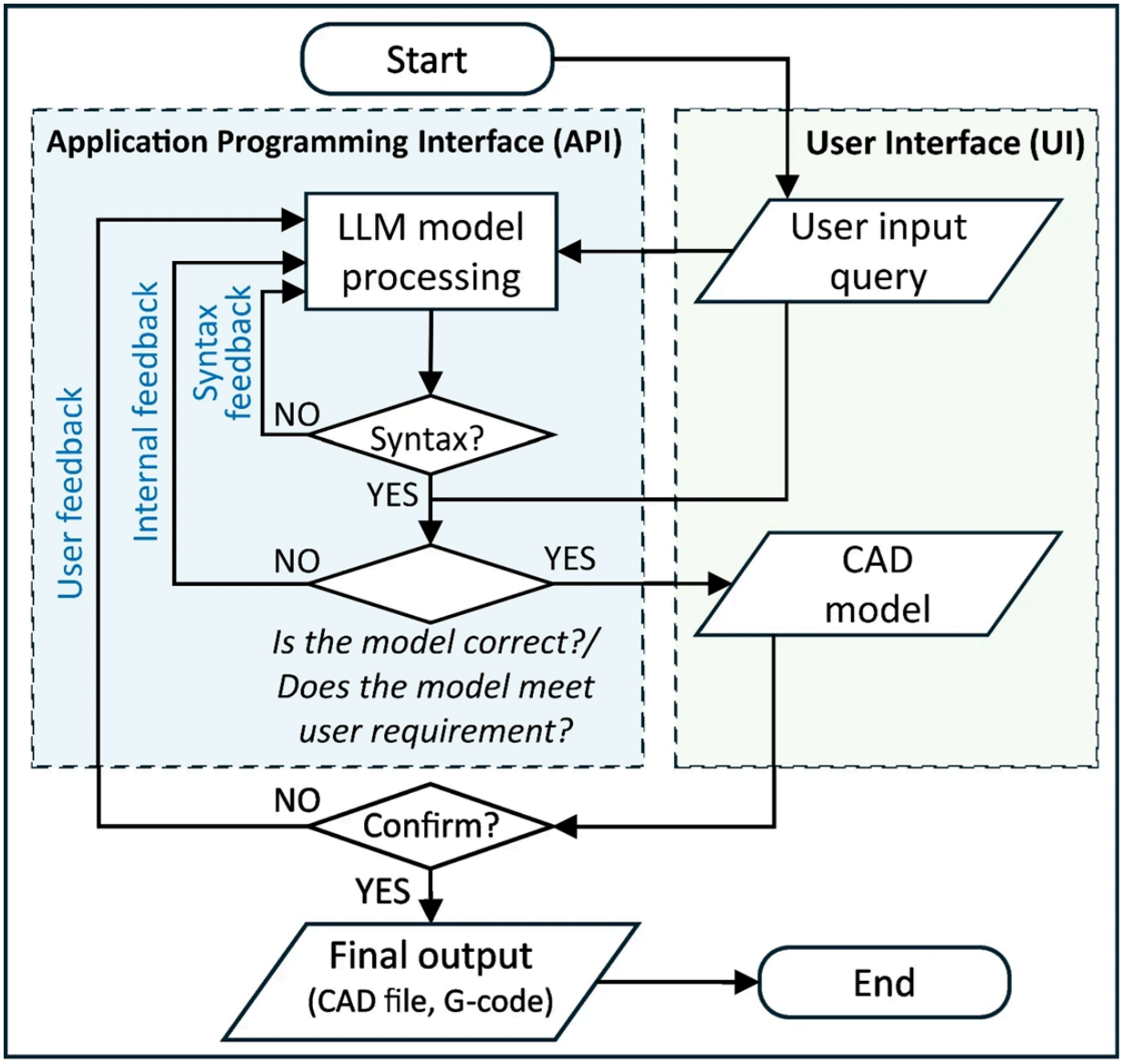

Figure: Example architecture of a text-to-CAD system (adapted from the Query2CAD research). A natural-language query is sent to an LLM (e.g. GPT-4) which generates a FreeCAD Python script. The script is executed to produce a 3D model. An error-checking module (or vision-based model) evaluates the result: if needed, the prompt is refined and the LLM is queried again. This closed loop continues until the design meets the user's intent.

Proposed Architecture

Building on these ideas, we propose a high-level architecture for our Text-to-CAD co-pilot (Figure above). The core components are:

- **User Interface (UI)**: A chat or command panel integrated into FreeCAD (or a companion app). The user enters natural-language instructions like *"Extrude a 10 mm circle by*

5 mm" or *"Drill a hole of radius 2 mm through the top face"*. The UI sends this query to the backend engine. This interface can be a dockable Qt/PySide panel inside FreeCAD, or a web-based interface (e.g. Streamlit or Electron) that communicates with FreeCAD via an API.

- **LLM Prompting Module**: The input text is packaged into a structured prompt for the LLM. The prompt includes context (e.g. "Use FreeCAD Python syntax") and any relevant constraints. We may use a prompt template such as: *"Generate a FreeCAD Python script that [does X]. Return only the code."* This prompt is sent to an LLM (via an API call or local model). For flexibility and privacy, the system can support either cloud-based APIs (OpenAI GPT-4) or open-source models (e.g. LLaMA, Mistral, or GPT4All) running locally or on a private server. To manage conversation context and iteration, a tool like LangChain can chain prompts and maintain history in memory.

CAD Engine (FreeCAD): The generated Python script is executed in FreeCAD. FreeCAD's Python API allows dynamic creation and modification of geometry. For instance, a script might look like this (illustrative example):

```
 import FreeCAD
doc = FreeCAD.newDocument()
cyl = doc.addObject("Part::Cylinder", "cyl")
cyl.Radius = 0.01  # 10 mm
cyl.Height = 0.1   # 100 mm
doc.recompute()
```

- This adds a cylinder (radius 10 mm, height 100 mm) to the new document. (FreeCAD examples show that "Functionalities of FreeCAD are accessible to Python and can be used to define geometrical models in simple Python scripts".) The script can include multiple commands, sketches, Boolean operations, and so on. Once executed, FreeCAD updates the 3D view with the new geometry.

- **Error Handling and Verification**: Immediately after execution, the system checks for errors. At minimum, we catch Python exceptions or execution errors (e.g. missing objects). We also verify geometry validity: for example, we may ensure Boolean operations succeed and no invalid solids are produced. If the script throws an error or generates an undesired model, the system **loops back to the LLM**. We construct an *error-driven prompt* that includes the original user instruction, the failed script, and the error message or a description of the flaw. The LLM is re-queried to fix the script or try a different approach (as in an iterative refinement loop). Optionally, a vision-language model can inspect a rendered image: e.g. a caption model or a Visual Question Answering (VQA) model can compare the outcome with the user's intent and suggest corrections, as done in Query2CAD.

- **Model Hosting / Cloud**: For computationally heavy LLM inference or to support team collaboration, parts of the system may run in the cloud. For example, the LLM and refinement loop can be hosted on a server (AWS, Azure, GCP) with GPU resources, while FreeCAD itself can run on a cloud instance or on the user's machine. The CAD models created can be stored centrally (e.g. in a shared repository or cloud storage) so multiple engineers can access and continue editing designs. The above-cited research explicitly notes that "future directions include integrating cloud-based execution" to streamline CAD automation. Our design allows flexible deployment: an organization could host the LLM and FreeCAD on-premises for sensitive projects, or use public APIs and storage for open designs.

- **Feedback to User**: Finally, the system presents the result to the user. The new or modified 3D model appears in the FreeCAD view. The generated script can also be shown (for transparency or manual tweaks). If an iteration loop occurred, the UI may inform the user of any changes from the original plan. The user can then accept the model or issue further commands (e.g. "add a chamfer" or "make it hollow"), continuing the interactive dialogue.

Technology Stack (Open Source)

Our prototype will use **FreeCAD** (an open-source parametric CAD software) as the modeling engine. FreeCAD offers a rich Python API for geometry creation. Key libraries and tools include:

FreeCAD 0.20/0.21: The core CAD application. We run it in either GUI mode (for integrated UI) or headless mode (for server execution). Scripting in FreeCAD uses `doc.addObject(...)` to create primitives (boxes, cylinders, spheres, etc.) and operations (boolean, pad, revolve). For example, a cylinder is created by:

```
 from FreeCAD import Base
doc = FreeCAD.newDocument()
cyl = doc.addObject("Part::Cylinder", "cyl")
cyl.Radius = 5  # in millimeters
cyl.Height = 10
doc.recompute()
```

- This corresponds to the FreeCAD documentation example.

- **Python 3.x**: The system glue. A Python application will orchestrate the UI, LLM calls, and FreeCAD commands. We use packages like `freecad` or `pyfreecad` to embed or automate FreeCAD. Prompt templates and chat logic can be managed with tools like **LangChain**, which helps structure multi-step prompts and store intermediate context.

- **LLM Backend**: We plan to support both **cloud LLM APIs** and **local open models**. For cloud, the OpenAI GPT-4 API (or GPT-4o) can be used as it excels at code generation. (This aligns with state-of-art work using GPT-4.) For an open-source alternative, we can use meta's LLaMA/Mistral-family models (via HuggingFace or Together AI) that can run on local GPU or CPU. The system will treat the model choice as a parameter: e.g. user can select GPT-4, GPT-4o, LLaMA-3, etc. In Query2CAD's code, they even allow `--code_gen_model codellama/chatgpt/llama3/gpt4-turbo`.

- **Free/Open CAD Tools**: While FreeCAD is our primary CAD kernel, we may leverage **CadQuery** (a Pythonic CAD library built on top of FreeCAD) for certain parametric constructs. CadQuery allows more concise Python code and could simplify some LLM prompts (e.g. "import cadquery as cq; model = cq.Workplane('XY').box(10,5,3)"). Similarly, **OpenSCAD** (script-based CAD) is used in GAD; it is easy to run and verify programmatically. We design the system modularly so that other kernels (OpenSCAD, Blender) could be plugged in later.

- **User Interface Framework**: For the CAD-integrated UI, we can build a **PySide2/Qt** panel inside FreeCAD that hosts a text entry and chat display. Alternatively, a lightweight web interface (built with **Streamlit** or **Flask**) can send commands via FreeCAD's Python console. The choice depends on user preference; the cited GAD system uses a browser UI (Streamlit) for accessibility.

- **Cloud Storage and Services**: 3D models (e.g. STEP or FreeCAD documents) will be saved to a cloud repository or shared network drive. We could use AWS S3 or a Git-based approach for versioning CAD files. If using a cloud LLM, secure API handling and compliance (SOC-2, encryption) must be considered. The architecture explicitly accommodates cloud or on-premises deployment (as recommended in recent research).

- **Error-Checking Libraries**: To verify CAD geometry, we may employ tools that analyze B-Rep validity. FreeCAD itself provides methods to check shapes; we will also rely on exception handling. In some designs, we might use computer vision libraries (e.g. PyTorch for VQA) if we incorporate image-based feedback loops (as seen in Query2CAD).

Example Workflow and Code

Here is a simplified example of how a text command can flow through the system:

1. **User prompt (text):** "Create a 20 mm wide by 10 mm thick base plate and extrude 100 mm upward."

LLM prompt construction: The system builds a prompt such as:

"Generate a FreeCAD Python script: Create a sketch of a 20 mm by 10 mm rectangle on the XY plane, then extrude it by 100 mm upward. Return only the code."

2.

LLM generates code: Using the GPT API or local model, we might get:

```
import FreeCAD, Part
doc = FreeCAD.newDocument("SketchPlate")
box = doc.addObject("Part::Box", "BasePlate")
box.Length = 20
box.Width = 10
box.Height = 100
doc.recompute()
```

3. (This example is consistent with FreeCAD scripting examples.)

4. **CAD execution:** The script runs in FreeCAD, creating a rectangular prism as requested.

5. **Error check:** No errors occur, so the 3D model is shown. If an error had happened (e.g. a syntax mistake), the system would catch it and ask the LLM to fix it by appending the error context to the prompt (prompt refinement).

6. **User confirmation:** The model appears in the CAD view. The user can accept or issue another command (e.g. “Drill a 5 mm hole through the center”).

Throughout this workflow, citations from the literature guide our design choices: using FreeCAD scripts as the output format, structuring a feedback loop, and employing open-source components (FreeCAD, Python, LLM libraries) to ensure extensibility.

Conclusion

A Text-to-CAD co-pilot promises to democratize 3D design by letting engineers converse with CAD tools in plain English. By combining cutting-edge LLMs with established CAD kernels like FreeCAD, the system can automate routine modeling tasks and speed up prototyping. The proposed architecture—LLM prompt ➔ script generation ➔ CAD execution ➔ error-feedback loop—is grounded in recent research and leverages open-source technology. Mechanical engineers gain an AI assistant that reduces scripting overhead, while designers can iterate more rapidly. As LLM capabilities improve and cloud infrastructure matures, such Text-to-CAD systems will become an integral part of the CAD workflow, paving the way for more natural and efficient design processes.

Sources: This document is based on recent research and tools in AI-driven CAD, including academic studies on LLM-CAD integration and open-source frameworks like Query2CAD and GAD. Example code follows FreeCAD documentation and tutorials. All cited sources are from peer-reviewed or authoritative publications in the field.