CADDesigner: Conceptual CAD Model Generation with a General-Purpose Agent

Fengxiao Fan*, Jingzhe Ni*, Xiaolong Yin, Sirui Wang, Xingyu Lu, Qiang Zou, Ruofeng Tong, Min Tang and Peng Du**

School of Computer Science and Technology, Zhejiang University, Hangzhou, China

ARTICLE INFO

Keywords:
CAD code generation
ReAct agent
Multimodal input
Large Language Models (LLMs)

ABSTRACT

Computer-Aided Design (CAD) is widely used for conceptual design and parametric 3D modeling, but typically requires a high level of expertise from designers. To lower the entry barrier and facilitate early-stage CAD modeling, we present CADDesigner, an LLM-powered agent for conceptual CAD design. The agent accepts both textual descriptions and sketches as input, engaging in interactive dialogue with users to refine and clarify design requirements through comprehensive requirement analysis. Built upon a novel Explicit Context Imperative Paradigm (ECIP), the agent generates high-quality CAD modeling code. During the generation process, the agent incorporates iterative visual feedback to improve model quality. Generated design cases can be stored in a structured knowledge base, providing a mechanism for continual knowledge accumulation and future improvement of code generation. Experimental results show that CADDesigner achieves competitive performance and outperforms representative baselines on conceptual CAD model generation tasks.

1. Introduction

Conceptual design of Computer-Aided Design (CAD) models often begins with incomplete or abstract specifications, requiring designers to translate rough ideas into precise parametric models. Creating such 3D models using traditional CAD software typically demands substantial expertise and familiarity with complex modeling operations, which can be a barrier for novice users and slow down early-stage product development. Traditional CAD platforms–such as OnShape, AutoCAD, SolidWorks, and CATIA–primarily rely on manual sketching and extrusion workflows, making iterative prototyping time-consuming and error-prone. However, recent advances in artificial intelligence, particularly the emergence of LLMs, present promising opportunities to automate CAD model generation, thereby reducing the entry barrier and accelerating the design workflow.

Early research on automatic CAD model generation has primarily focused on parametric modeling approaches [1, 2, 3, 4]. However, these methods are constrained by the limited diversity of training data and the representational capacity of their model outputs, supporting only a narrow set of CAD operations. Recent work leveraging LLMs for CAD code generation has shown promise in overcoming these limitations [5, 6, 7]. Fine-tuned LLMs can interpret multimodal user inputs and generate corresponding CAD modeling code. Nevertheless, fine-tuning open-source LLMs requires substantial GPU resources, and high-quality CAD training datasets remain scarce, limiting the diversity and quality of the generated code.

To address these challenges, we propose CADDesigner, an LLM-powered agent for conceptual CAD design. CADDesigner accepts both textual descriptions and sketches as input and engages in interactive dialogue with users to refine design requirements. The system combines knowledge-constrained code generation with iterative visual feedback, enabling the agent to generate high-quality CAD modeling code that better aligns with user design intent.

To further improve code generation quality, we propose the Explicit Context Imperative Paradigm (ECIP), an LLM-oriented representation style implemented through a lightweight API layer built on top of CadQuery [8]. ECIP explicitly represents modeling context and restructures CAD operations into an explicit imperative workflow, enabling clearer state transitions, more reliable code generation, and easier iterative correction while remaining aligned with CadQuery semantics. Representative results generated by CADDesigner are shown in Figure 1, and our main contributions are summarized as follows:

- We present CADDesigner, a framework for conceptual CAD model generation. CADDesigner accepts textual descriptions and sketches as user input, and integrates tools for requirement analysis, knowledge-constrained code generation, and vision-based error correction to generate CAD scripts that satisfy user intent.
- We introduce an explicit context imperative paradigm for CAD modeling script generation based on an imperative function-calling structure. This paradigm decouples individual CAD operations and improves code generation quality through explicit type annotations, LLM-friendly error representations, and self-evolving capabilities. It supports a wide range of operations, including extrusion, revolution, fillet, chamfer, sweeping, lofting, etc.

The remainder of this paper is organized as follows. We begin with a review of related work. We then introduce our proposed CAD modeling code generation framework, CADDesigner, along with the novel code generation paradigm designed to enhance code quality. Next, we describe our

*Equal contributions.
**Corresponding author: dp@zju.edu.cn

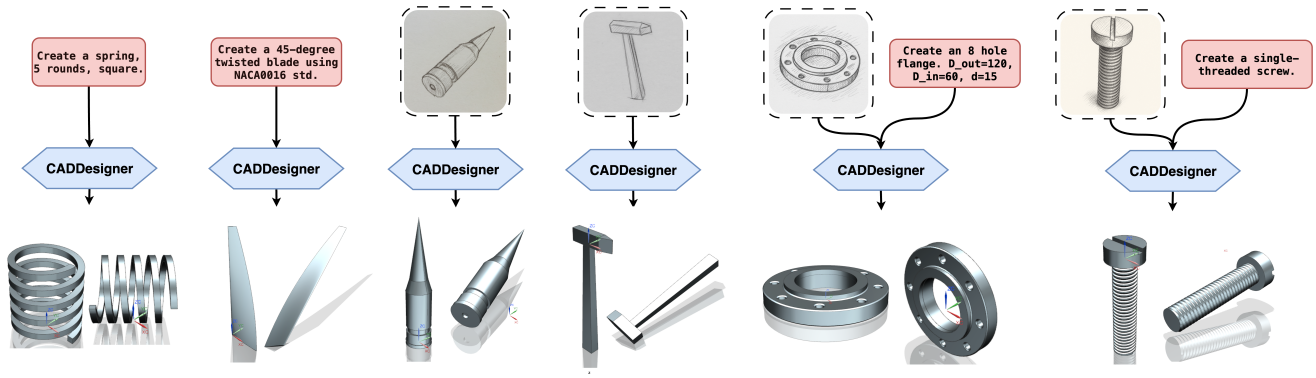

Figure 1: Demonstration of various CAD models generated by CADDesigner. Our method supports multimodal input and a broad range of CAD operations, including extrusion, revolution, fillet/chamfer, sweeping, lofting, etc., as well as the creation of standard components such as flanges and screws.

experimental setup and present the results. Finally, we provide a comprehensive analysis and comparison to highlight the strengths and limitations of the proposed approach.

2. Related Work

This section provides a comprehensive review of four key areas relevant to our work: parametric CAD modeling, LLM-driven CAD generation, agent-based model generation, and parametric CAD modeling SDKs.

2.1. Parametric CAD Model Generation

Parametric CAD model generation approaches commonly employ supervised learning techniques to produce sequences of CAD commands, which can be imported into CAD modeling software to generate editable, parametric files [9, 1]. Systems such as DeepCAD [1], Fusion 360 Gallery [9], SkexGen [10], and Diffusion-CAD [11] adopt customized neural network architectures trained to generate command sequences. However, they primarily support only basic modeling operations such as sketching and extrusion, and struggle to generalize to more complex modeling procedures.

Additional efforts [3, 12, 13] explore CAD model reconstruction from point cloud data, aiming to recover structured geometry from unorganized 3D observations. More recent methods [14, 15] further attempt to infer structured, editable CAD models directly from unstructured RGB images (either single- or multi-view), bridging the gap between 2D visual understanding and 3D CAD modeling. Despite these advancements, existing approaches remain limited in their ability to generate complex CAD models, primarily due to constrained training data diversity and limited output representation capacity.

2.2. LLM-based CAD Model Generation

Fine-tuned LLMs and vision-language models (VLMs) have emerged as a prominent direction for CAD model generation, enabling the synthesis of CAD command sequences or Python code from multimodal input. Cad-LLM [16] and CadVLM [17] utilize fine-tuned LLMs to generate parametric models from sketches and images. CAD-MLLM [18] targets multimodal inputs–including text, images, and point clouds–for parametric model generation. CAD-Llama [5] proposes a hierarchical annotation pipeline that transforms command sequences into semantically structured CAD code. This is followed by adaptive pre-training and instruction tuning, enabling LLMs to generate high-fidelity parametric 3D models. CAD-Recode [7] converts point clouds into sequential representations, then uses a pre-trained LLM (Qwen2-1.5B [19]) to generate corresponding CAD code. Text-to-CadQuery [20] directly maps text descriptions to CadQuery code, leveraging pre-trained LLMs' capabilities in Python generation and spatial reasoning.

While these methods demonstrate strong generation capabilities, they remain limited by several factors: the computational cost of fine-tuning large models, the closed nature of many commercial LLMs, and the scarcity of high-quality, diverse training datasets for CAD.

2.3. Agent-based CAD Model Generation

Another research direction focuses on constructing CAD generation agents using prompt engineering and ultra-large foundation models, circumventing the need for additional training. CAD-Assistant [21] presents a general-purpose CAD agent framework that integrates visual and language models (VLLMs) as planners to generate Python code for FreeCAD [22], achieving zero-shot capability across diverse CAD tasks. SeekCAD [23] combines retrieval-augmented generation (RAG), visual feedback, and a chain-of-thought (CoT) mechanism to generate customized CAD modeling code with self-optimization. 3D-PreMise [24] investigates the potential and limitations of LLMs in program synthesis for manipulating 3D software and generating parametrically controlled shapes. CADCodeVerify [25] incorporates iterative validation and refinement loops, using visual language models to pose and answer verification questions about the generated CAD outputs.

In contrast to these works, CADDesigner accepts multimodal user inputs–text and sketches–to capture detailed

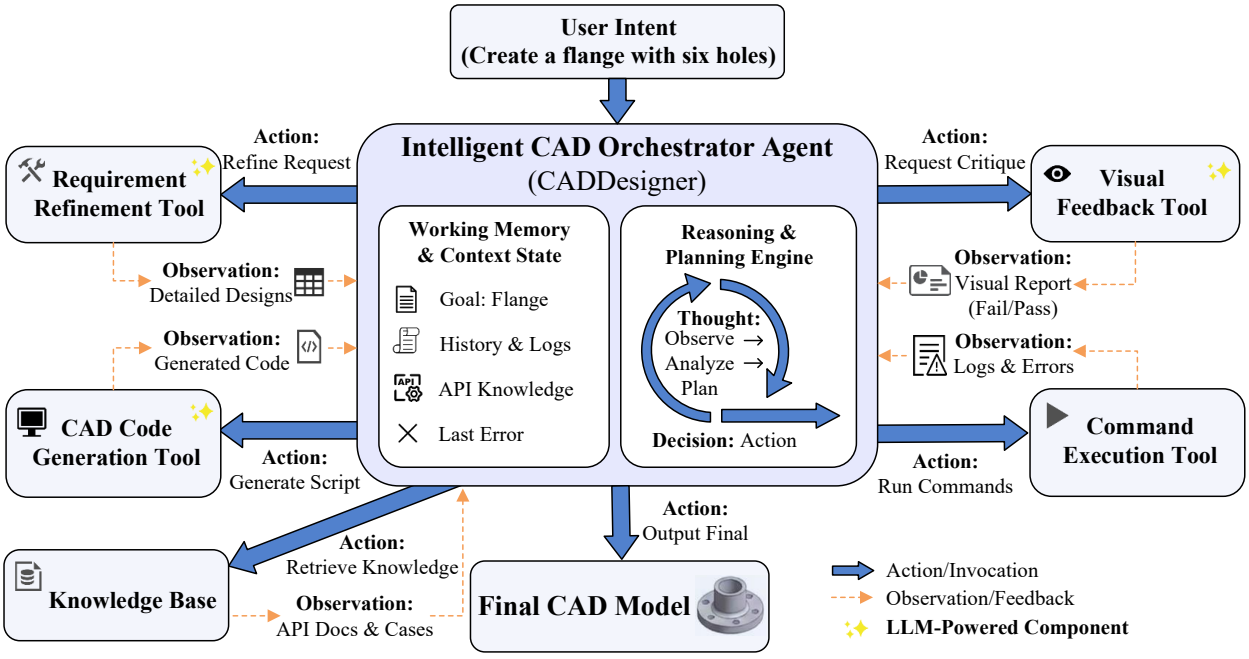

Figure 2: The Intelligent CAD Orchestrator Agent, CADDesigner, follows a ReAct-style paradigm to progressively transform user requirements into valid CAD models through iterative reasoning, tool execution, and feedback refinement. It first refines user requirements into detailed designs, generates executable modeling code using domain APIs, and analyzes execution results via both symbolic (e.g., shell logs and errors) and visual feedback (e.g., rendered 3D views).

design requirements, engages in interactive refinement with users, and enhances code quality and model fidelity through visual feedback and iterative optimization.

2.4. Parametric CAD Modeling SDKs

Beyond modeling methods, existing CAD modeling SDKs play a crucial role as the underlying representations for code-based CAD generation. Most Python-based parametric CAD frameworks are built upon OpenCASCADE Technology (OCCT), a widely used boundary representation modeling kernel. CadQuery provides a high-level Python interface with a Fluent API based on method chaining and workplane abstractions [8], enabling concise modeling workflows through implicit context propagation. Build123d adopts a different design based on context managers (e.g., `with` statements) and operator-style modeling by overloading operators in Python, offering tighter integration with native Python constructs and improved modularity [26].

However, these SDKs are not specifically designed for LLM-based code generation. CadQuery's Fluent API relies on implicit internal state transitions, which can obscure intermediate modeling context and increase the difficulty for LLMs to track dependencies across operations. Despite its explicit context declaration, build123d's reliance on symbolic operators (@, +, −) for complex modeling may make semantic intent less transparent for LLM-based code generation, especially compared with explicit function-call interfaces. Such implicit context handling and lack of well-defined semantics can lead to ambiguity and reduced reliability in generated code. These limitations motivate the need for a more explicit and LLM-friendly CAD representation, which we address through the proposed Explicit Context Imperative Paradigm (ECIP).

3. CAD Modeling Architecture

In this section, we introduce the CADDesigner architecture, which features a ReAct-style agent interacting with a diverse set of integrated CAD tools to iteratively generate and refine CAD models. The overall framework is illustrated in Figure 2.

3.1. ReAct Agent rather than Workflow

CADDesigner employs a ReAct-style [27] agent-centric architecture, where a central agent governs the entire modeling loop through iterative reasoning, tool execution, and feedback integration.

Unlike traditional hard-coded workflows, our architecture leverages prompt-based workflow specifications, where the system prompts encode procedural guidance for the agent. This design allows for flexible and revisable task orchestration, as the workflow is not statically programmed but interpreted at runtime by the agent. As a result, users can intervene at any stage of the process, like adding constraints, correcting directions, or adjusting goals, leading to a truly interruptible and human-in-the-loop modeling cycle.

The ReAct-style loop includes reasoning, acting, and reflecting: the agent expands coarse inputs into parameterized specs, generates executable code, renders outputs, evaluates results, and iteratively refines the model until user intent is satisfied. By consolidating control within the agent, we

enable adaptive tool use and interpretable, interactive CAD automation in under-specified or evolving design tasks.

3.2. CAD Tools

The intelligent orchestration relies on a diverse suite of CAD-oriented tools. Each tool addresses a specific stage within the design-to-modeling pipeline and works in synergy under the direction of the ReAct agent. This subsection describes the CAD tools integrated in our pipeline.

3.2.1. Core CAD Toolset

CADDesigner employs various CAD-specific tools to support its design-to-modeling pipeline. Central to this set are four tools, denoted $\mathcal{T} = \{T_1, T_2, T_3, T_4\}$, each fulfilling a distinct role.

- **Requirement Refinement Tool** $T_1 : \mathcal{I} \rightarrow \mathcal{D}$ maps user-provided multimodal input $\mathcal{I}$ to detailed design descriptions $\mathcal{D}$, refining initial requirements into structured design specifications.
- **Code Generation Tool** $T_2 : \mathcal{D} \rightarrow \mathcal{C}$ translates finalized design descriptions into executable CAD modeling code $\mathcal{C}$. A new code paradigm for CAD code generation has been designed and used by T_2. This is discussed in detail in Section 4.
- **Command Execution Tool** $T_3 : \mathcal{C} \rightarrow \mathcal{M}$ executes the generated Python code $\mathcal{C}$ to produce the corresponding CAD model $\mathcal{M}$ by running shell commands.
- **Visual Feedback Tool** $T_4 : \{\mathcal{M}, \mathcal{D}\} \rightarrow (P, F)$ outputs both a high-level termination signal $P \in \{0, 1\}$, where $P = 1$ triggers termination, and structured diagnostic feedback F for refinement. Inside this tool, a multi-view rendering result of model $\mathcal{M}$ will be generated. We denote this render result as $\mathcal{V}$. Thus T_4 can be separated into 2 steps: $T_4^1 : \mathcal{M} \rightarrow \mathcal{V}$, and $T_4^2 : \{\mathcal{V}, \mathcal{D}\} \rightarrow (P, F)$

The sequential composition of these core tools forms the main functional pipeline:

$$T_4^1 \circ T_3 \circ T_2 \circ T_1 : \mathcal{I} \rightarrow \mathcal{V}. \tag{1}$$

This pipeline is augmented with a closed-loop correction mechanism: after generating visual feedback $\mathcal{V}$, the system invokes T_4^2 to obtain (P, F). If $P = 0$, the feedback F is used to guide revised design generation via $T_3 \circ T_2 \circ T_1$. If $P = 1$, the process terminates successfully. Otherwise, the loop continues until T_4^2 returns a termination signal.

3.2.2. Auxiliary Toolset

In addition to the core CAD toolset, CADDesigner integrates several auxiliary tools that support efficient context management and system interaction. These include:

- **SketchPad Tool**: A flexible key-value storage tool designed to mitigate the context explosion problem in LLM-based agents. SketchPad allows the agent to store arbitrary intermediate data at any time during the design process, including image paths, reference code snippets, execution results, and more. By storing context data as keyed entries instead of embedding entire data repeatedly, it significantly reduces LLM context size. SketchPad also supports automatic summarization, tag-based search, and automatic expiration of outdated entries, enabling flexible and scalable context management.
- **File Operation Tools**: These tools provide fundamental file-reading and file-writing capabilities, allowing the agent to store or load files as required throughout the CAD design pipeline. Similar to SketchPad, file content or references are stored and accessed efficiently to avoid large context overheads.

Together, these auxiliary tools complement the core CAD toolset by enabling robust, efficient, and scalable operation of the CADDesigner agent in complex multi-turn interactions.

3.3. Requirements Analysis

During the conceptual design phase, designers typically have only rough descriptions or sketches, making it challenging to precisely define detailed model parameters as well as the modeling process. To address this, we support designers by conducting requirement analysis to clarify and refine their design intention.

Let the set of initial information provided by the user be $\mathcal{I} \subseteq \mathcal{I}_{\text{txt}} \times \mathcal{I}_{\text{img}}$, where $\mathcal{I}_{\text{txt}}$ denotes the set of possible text inputs (including empty text $\emptyset$) and $\mathcal{I}_{\text{img}}$ denotes the set of possible image inputs (including a null image $\emptyset$). Let $\mathcal{P}$ be a set of parameters sufficient for generating a CAD model for the current design task. The process by which CADDesigner conducts detailed design can be expressed as $\mathcal{F}_{\text{design}} : \mathcal{I} \rightarrow \mathcal{D}_{\text{detail}}$, where $\mathcal{D}_{\text{detail}}$ denotes the detailed design containing specific parameters, satisfying $\mathcal{P} \subseteq \mathcal{D}_{\text{detail}}$, and this process is accomplished by T_1. At this stage, CADDesigner enables users from different disciplines to realize their design intentions through interaction. The user's revision process for the detailed design is modeled as an iterative function:

$$R : \mathcal{D}_{\text{detail}} \times \mathcal{U} \rightarrow \mathcal{D}_{\text{detail}}, \tag{2}$$

where $\mathcal{U}$ is the set of user revision operations. After n iterations of revision, we obtain $\mathcal{D}_{\text{final}} = R^n(\mathcal{D}_{\text{detail}}, \mathcal{U})$, which is then passed to the next stage.

3.4. Knowledge Constrained Code Generation

$\mathcal{D}_{\text{final}}$ is the detailed design confirmed by the user. The process by which the code generation tool T_2 receives $\mathcal{D}_{\text{final}}$ and generates CAD modeling code is expressed as $G : \mathcal{D}_{\text{final}} \rightarrow \mathcal{C}$. A structured knowledge base $\mathcal{K}$ is constructed to support code generation. $\mathcal{K}$ includes a set of function annotations $Anno \subseteq \mathcal{K}$ and a set of basic model cases $Case \subseteq \mathcal{K}$, which provide semantic and structural references for the code generation tool. With the assistance of the knowledge base, the code generation process is extended to:

$$G' : \mathcal{D}_{\text{final}} \times \mathcal{K} \rightarrow \mathcal{C}', \tag{3}$$

where C' denotes the code generated by referencing the knowledge base, which is semantically and structurally richer compared to the code C generated without using the knowledge base. The execution result of the code, including detailed geometry metadata (see Section 4.2.1, Metadata & Tagging) such as volumes and measurements of some parts of the model, is returned after execution. If the execution succeeds, the code can run successfully and generate a model $\mathcal{M} = T_3(C')$, which is then passed to the next stage; if the execution fails, structured error information is returned to guide the next iteration of code generation G' to produce new code. This feedback loop helps to iteratively correct the code and improve the success rate. Therefore, the knowledge base provides essential semantic and structural support for code generation, and the iterative process combined with execution feedback further enhances the likelihood of successful code generation. In addition, execution-time geometric metadata and queryable CAD-native representations provide structured signals for dimension-sensitive and constraint-related verification beyond visual inspection.

3.5. Vision-based Iterative Error Correction

Multi-view snapshots of the generated CAD model $\mathcal{M}$ are taken by T_4^1, producing an image set $\mathcal{V} = \{\text{Img}_1, \text{Img}_2, \dots, \text{Img}_6\}$ (corresponding to the Front-Top-Left, Back-Bottom-Right, Back-Top-Left, Front-Bottom-Right, Top, and Right views respectively). CADDesigner evaluates whether the model meets the user's intent $\mathcal{D}_{\text{final}}$ by using both a high-level termination signal $P \in \{0, 1\}$ and structured diagnostic feedback generated from multi-view observations.

Internally, P is determined by T_4^2, which involves two components:

- **Visual Question Generation** $F_{\text{vq}} : \mathcal{V} \times \mathcal{D}_{\text{final}} \to \mathcal{Q}$, which produces a set of targeted visual questions $\mathcal{Q} = \{q_1, q_2, \dots, q_n\}$ based on the user's requirements and the multi-view images $\mathcal{V}$.
- **Visual Feedback Generation** $F_{\text{vf}} : \mathcal{V} \times \mathcal{Q} \to (P, F)$, which analyzes the multi-view images $\mathcal{V}$ to answer the questions in $\mathcal{Q}$, and outputs explicit visual feedback F along with the binary decision P. When $P = 0$, it returns visual feedback F to the code generation stage to regenerate the code; when $P = 1$, the result is delivered to the user for final judgment. The user's satisfaction with the modeling result is determined by $Sat : \mathcal{M} \to \{0, 1\}$. If $Sat(\mathcal{M}) = 1$, $\mathcal{M}$ is added to the case library $\mathcal{K}$, that is, $Case' = Case \cup \{\mathcal{M}\}$; if $Sat(\mathcal{M}) = 0$, it re-enters the code generation stage.

In the entire process, except for the user feedback links (R and Sat), all state transitions $Trans : State \to State'$ are determined independently by CADDesigner.

4. CAD Code Generation Paradigm

In this section, we provide a detailed description of a new CAD code generation paradigm from four aspects: the Explicit Context Imperative Paradigm (ECIP), its practical realization via an ECIP-based CAD API, LLM-friendly CAD code design, and the construction of a reusable modeling knowledge base.

4.1. Explicit Context Imperative Paradigm

In recent years, many works have employed **CadQuery** (CQ) code as an intermediate representation for CAD model generation. CadQuery commonly employs a Fluent API design paradigm [8], where operations are expressed via chained method calls to enable a highly readable and compact modeling workflow. This style relies on an *implicit internal context* C that tracks the current modeling state and is automatically passed between operations. Formally, each operation f_k updates the hidden state C_k with parameters p_k:

$$C_{k+1} = f_k(C_k, p_k). \tag{4}$$

A chained expression can therefore be written as:

$$C_n = f_n \circ f_{n-1} \circ \cdots \circ f_1(C_0). \tag{5}$$

While concise, this implicit state can make semantic dependencies less explicit, which may complicate automated code generation for LLMs that must track intermediate contexts. In contrast, low-level APIs like PythonOCC require explicit inputs for each operation, yielding clear semantics but verbose, fine-grained code that lacks high-level abstraction and usability.

To address these limitations, we introduce the **Explicit Context Imperative Paradigm (ECIP)**, which enforces explicit passing of all input objects and parameters to each operation. Formally, each operation g_k maps explicit input state S_k and parameters p_k to a new state S_{k+1}:

$$S_{k+1} = g_k(S_k, p_k). \tag{6}$$

Thus, the modeling workflow is represented as a sequence of explicit state transformations:

$$S_1 = g_1(S_0, p_1),\ S_2 = g_2(S_1, p_2),\ \dots,\ S_n = g_n(S_{n-1}, p_n), \tag{7}$$

where all intermediate states S_k are explicit variables.

Figure 3 illustrates a typical code comparison between CadQuery and ECIP. CadQuery (left) uses chained method calls that implicitly pass context, making the code compact but less flexible for inserting standard Python statements or loops. In contrast, ECIP (right) explicitly passes all inputs, allowing the use of standard Python control flow like `for` loops and variable assignments, which enhances modularity, readability, and ease of debugging. Thus, ECIP combines the clarity of low-level APIs with the expressiveness of high-level modeling, improving the accuracy and interpretability of agent-driven CAD code generation. This does not imply that CadQuery cannot support more explicit coding styles; rather, ECIP standardizes such explicitness in a form that is more suitable for LLM-oriented generation and refinement.

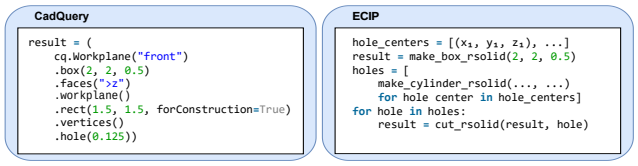

Figure 3: Code comparison between CadQuery (left) and ECIP (right). ECIP explicitly passes context and supports standard Python constructs, improving code clarity and flexibility.

4.2. ECIP-Compliant CAD API Design

ECIP (referred to as SimpleCADAPI in the actual project) is designed as a command-style Python API built on top of CadQuery's `occ_impl.shapes` module. It serves as an LLM-oriented intermediate representation that remains semantically aligned with CadQuery. The geometric classes (e.g., `Vertex`, `Edge`, `Solid`) are implemented as lightweight wrappers over CadQuery objects, preserving full interoperability with native CadQuery operations.

The design follows the Open-Closed Principle: core geometric types are closed for modification, while operations and extensions are introduced through additional functions. Moreover, SimpleCADAPI is fully compatible with the NumPy library, enabling efficient mathematical computation and geometric transformation directly on CAD entities. It provides explicit input-output interfaces for each operation, facilitating LLM-driven code generation, iterative correction, and modular composition, while retaining the high-level modeling capabilities of CadQuery. SimpleCADAPI is organized into two main parts: the **Core** module and the **API** module.

4.2.1. Core Module

The Core module encapsulates fundamental geometric types and primitive operations. By wrapping the underlying OpenCASCADE `TopoDS_Shape` classes, it establishes a clear hierarchy and semantic structure for geometric objects, while also managing coordinate systems, supporting tagging and metadata attachment, and enforcing strict type annotations and exception handling.

- **Geometric Object Hierarchy:** The Core module defines a layered geometric object model comprising the following primary classes, all inheriting from `TaggedMixin` to provide unified tagging and metadata management:

  (1) Vertex: 0-dimensional points in 3D space.

  (2) Edge: 1-dimensional curves connecting vertices.

  (3) Wire: Ordered collections of edges forming open or closed chains.

  (4) Face: 2-dimensional bounded surfaces defined by wires (outer boundaries and holes).

  (5) Shell: Collections of faces forming partial or complete shells.

  (6) Solid: Closed 3-dimensional volumes with definable volumetric properties.

  (7) Compound: Arbitrary aggregates of geometric entities for grouping or assembly construction.

- **Coordinate Systems and Working Planes:** ECIP employs a unified coordinate system framework to ensure consistency and precision in geometric modeling and transformation. This framework includes:

  (1) CoordinateSystem: Represents a 3D spatial frame with an explicit origin, X-axis direction, and normal vector, providing an abstraction for spatial transformations.

  (2) WORLD_CS: A global singleton coordinate system instance defining the default reference frame. It follows the right-hand rule with origin at $[0, 0, 0]$, the X-axis pointing along the default width direction, the Y-axis along the default height direction, and the Z-axis upward.

  (3) SimpleWorkplane: Manages local coordinate systems via context managers, enabling nested coordinate frames with defined origins and orientations. The returned workplane object follows the `_wp` naming convention when the variable is not directly referenced within the block, ensuring linter and LLM compatibility. This simplifies relative geometric operations and supports complex modeling workflows.

- **Metadata & Tagging:** ECIP unifies semantic tagging and geometric metadata through a lightweight tag–metadata layer. Each entity maintains user-visible tags and structured metadata, while runtime lineage is stored separately to avoid polluting modeling data. This design supports three complementary mechanisms. First, primitive constructors perform geometry-driven tagging. For example, `make_box_rsolid` automatically assigns face-level semantic tags such as "top", "bottom", "left/right", and "side" based on face normals, edge structure, and primitive type. Second, derived operations attach semantic tags via topology tracking after geometric or topological edits. ECIP wraps OpenCASCADE builders and queries `Modified()`, `Generated()`, and `IsDeleted()` to compute a `TopoDelta`. Resulting faces are then automatically assigned operation-aware semantic tags (e.g., `origin.tool`, `op.cut.generated`, `origin.body`). This approach avoids brittle index-based bookkeeping and preserves semantic lineage across boolean and feature operations. Third, ECIP stores structured geometric metadata alongside shapes, including primitive type, box size (x, y, z), cylinder axis and height, sphere center and radius, coordinate system snapshots, topological references (`topo_ref`), and compact selector hints (e.g., center, normal, area, length, volume, and bounding box). These metadata are serializable and directly consumable by QL predicates such as `ql.tag(...)` and `ql.meta(...)`.

- **Query Language (QL) for Geometric Selection:** SimpleCADAPI provides a dedicated query language module (`ql`) for declarative geometric selection. QL selectors are fully serializable, enabling transparent logging and replay of selection operations. A selector is constructed by chaining entry-point functions (e.g., `ql.edges()`, `ql.faces()`),

Table 1
Summary of core CAD modeling functions by category

Category	Examples	Count
Creation	`make_box_rsolid`, `make_cylinder_rsolid`, `make_circle_rwire`	25
Transformation	`translate_shape`, `rotate_shape`, `mirror_shape`	3
3D Operations	`extrude_rsolid`, `revolve_rsolid`, `loft_rsolid`, `sweep_rsolid`	4
Boolean	`union_rsolidlist`, `cut_rsolidlist`, `intersect_rsolidlist`	3
Advanced Features	`fillet_rsolid`, `chamfer_rsolid`, `shell_rsolid`, `helical_sweep_rsolid`	4
Pattern	`linear_pattern_rsolidlist`, `radial_pattern_rsolidlist`	2
Tagging & Selection	`set_tag`, `ql.tag`, `ql.meta`, `ql.where`, `ql.order_by`	14
Export	`export_step`, `export_stl`	2

predicates (e.g., `ql.tag()`, `ql.meta()`), ordering keys (e.g., `ql.center_axis()`, `ql.geo()`), and cardinality constraints (e.g., `.take(n)`, `.exactly(n)`). Crucially, QL unifies two complementary selection modes: querying semantic lineage derived from `TopoDelta`-based tracking, and direct geometric selection based on shape properties. The example below illustrates both behaviors within a unified workflow.

- **Example Usage:**

```python
from simplecadapi import *

# Base solid with user tags and metadata
base = make_box_rsolid(width=12, height=8, depth=4)
base.add_tag("base_frame")
base.set_metadata("material", "aluminum")

# Add a boss in a local workplane
with SimpleWorkplane(origin=(0, 0, 4)) as _wp:
    boss = make_cylinder_rsolid(radius=2.5, height=2)
model = union_rsolidlist(base, boss)

# Through-hole cut (TopoDelta tags attached)
tool = make_cylinder_rsolid(radius=1.0, height=6)
model = cut_rsolidlist(model, tool)

# Semantic selection via topology tracking
new_cut_faces = (
    ql.faces().where(
        ql.and_(
            ql.tag("origin.tool"),
            ql.meta("track.event", "==", "generated"))
    ).resolve(model)
)
# Geometry-based selection (order-independent)
top_hole_edge = (
    ql.edges().where(ql.curve_type("circle"))
    .order_by(ql.center_axis("z"), desc=True)
    .take(1).exactly(1)
)
# Apply chamfer
result = chamfer_rsolid(model, top_hole_edge, distance=0.5)
```

4.2.2. API Module

The API module builds upon the Core module and provides higher-level composite operations and LLM-friendly interfaces for advanced modeling tasks. Its design mainly includes the following two aspects:

- **Core Modeling Functions**: A comprehensive set of over 50 CAD modeling functions organized into categories such as creation, transformation, 3D operations, boolean operations, advanced features, pattern generation, tagging & selection, and export. These functions allow users or agents to flexibly compose complex modeling procedures while maintaining explicit state transitions and clear semantics. For user-facing convenience, some primitive constructors expose semantic dimension names such as `width`, `height`, and `depth`, but the API documentation binds them explicitly to the world axes as x, y, and z, respectively. A summary of the function categories, representative examples, and counts is shown in Table 1.
- **Automation and Self-Maintenance**: To maintain consistency and support continuous evolution, the API module incorporates an automation pipeline with the following functions:
  - **(1) Source Code Parsing**: Automatically parses user-defined Python source files to extract and integrate new or updated modeling functions into the public API.
  - **(2) Function Categorization**: Scans and categorizes source code to generate up-to-date API initialization files and alias mappings for consistent public exposure.
  - **(3) Documentation Generation**: Extracts function signatures and docstrings to produce structured markdown documentation, improving usability and maintainability.
  - **(4) Knowledge Base Synchronization**: Synchronizes generated documentation with an external knowledge base, enabling keyword-based indexing and improved traceability.

4.3. LLM-Friendly CAD Code Design

Beyond the ECIP paradigm and its API implementation, we incorporate several complementary design features aimed

at improving LLM agents' ability to accurately generate and debug CAD modeling code.

4.3.1. Structured Error Information

The error handling mechanism of CadQuery typically produces generic or partially unstructured error messages, which can make automated debugging and code correction more challenging in some scenarios. To facilitate more effective guidance for LLM-driven code generation, ECIP introduces fine-grained structured error messages at the atomic operation level, represented as triples:

$$\texttt{ErrMsg} = (\texttt{ErrCau}, \texttt{ErrLoc}, \texttt{CorrAct}), \tag{8}$$

where `ErrCau` identifies the root cause of the failure, `ErrLoc` specifies the exact location of the error in the script, and `CorrAct` suggests potential corrective actions. This structured format provides actionable feedback that LLMs can leverage to improve fault localization and enable automated code repair.

4.3.2. Explicit Type Annotations

Type ambiguity commonly leads to erroneous code generation, particularly when dealing with closely related geometric entities such as Wire and Edge. ECIP mitigates this issue by enforcing explicit type annotations through a disciplined naming convention for functions, following the pattern:

$$\texttt{ActionName\_rReturnType}, \tag{9}$$

where `ActionName` denotes the CAD operation, `r` is a fixed prefix indicating "returns", and `ReturnType` explicitly declares the type of the returned object. This ensures that even without inspecting documentation, the function signature itself conveys type information, reducing ambiguity during LLM inference. Each function is also accompanied by comprehensive, type-annotated documentation. This explicit type information reduces ambiguity and improves code generation accuracy by LLMs.

4.3.3. Self-Evolving Composite Operations

To facilitate knowledge accumulation and abstraction, ECIP supports the definition of *composite operations*–higher-level constructs formed by encapsulating sequences of atomic operations. These composites are saved as new operations within the language, enabling agents to invoke them directly in subsequent tasks.

This self-evolving mechanism effectively expands the language's set of available operations, allowing reuse of complex modeling patterns such as screws, flanges, or other parametric features. In principle, this can improve one-shot generation success by allowing LLM agents to reuse accumulated domain knowledge more efficiently.

4.4. Knowledge Base for Code Framework

LLMs possess strong generative and reasoning abilities but often struggle with the fine-grained semantic requirements of CAD APIs, such as precise parameter formats, geometric constraints, and operation-specific return types. These limitations frequently lead to misinterpreted command signatures or hallucinated inputs, resulting in unstable code generation.

To provide explicit, domain-grounded guidance during modeling, we construct a structured knowledge base $\mathcal{K}$ coupled with retrieval-augmented generation (RAG). Retrieved function annotations and example cases supply accurate API semantics that complement the LLM's implicit priors, helping improve generation reliability. The knowledge base $\mathcal{K}$ consists of two key components:

- **Function Annotations** (`Anno`): Semantic descriptions of API functions, including parameter formats, return types, and usage notes;
- **Case Examples** (`Case`): Initially composed of manually selected examples and gradually expanded during use.

To automate the construction of $\mathcal{K}$, we design a rule-based pipeline that extracts and organizes information directly from the source code. Specifically, we collect function signatures, parameter types, and return types from Python annotations and type hints, and align them with the corresponding natural language descriptions provided in function-level docstrings. This process yields a structured and semantically consistent API documentation corpus.

4.4.1. Structured Representation for Retrieval

We further organize the extracted knowledge into a standardized documentation format to support RAG. Each API entry is represented as a structured Markdown document following a unified template, including function signatures with type annotations, purpose descriptions, parameter specifications, return values, possible exceptions, and practical usage examples. All entries are presented using second-level Markdown headings for structural consistency.

This standardized representation improves retrieval accuracy, reduces ambiguity in natural language queries, and enhances interpretability for both models and developers. For example, a CAD operation such as `chamfer_rsolid` specifies its input types (e.g., a `Solid` object, a list of `Edge` objects, and a `float` distance parameter) together with usage examples, enabling the retriever to match queries with correct API semantics and usage patterns.

4.4.2. Semantic-Friendly Chunking

When building the vectorized representation of $\mathcal{K}$, we employ a customized **semantic-friendly chunking** strategy based on the standardized Markdown structure of API documents. Specifically, each second-level heading (##) is treated as a boundary defining a semantic block. This design ensures **semantic completeness** (each chunk forms a self-contained unit such as a method or parameter description), **semantic independence** (different chunks can be processed and retrieved modularly), and **structural alignment** (the chunking respects the inherent document structure and avoids arbitrary or mid-sentence splits).

Compared to naive fixed-length chunking, this strategy preserves the logical integrity of API documentation and ensures that retrieved content remains coherent and directly usable during generation.

To further improve retrieval precision, we append **anchor keywords** to each chunk, which explicitly summarize its semantic context. These anchors consist of the API name, the section label, and a small set of representative content tokens. For example, a chunk corresponding to the purpose section of `chamfer_rsolid` may include:

[`chamfer_rsolid`, `purpose`, `chamfer`, `edge`, `transition`]

These anchors act as explicit semantic signals that enhance embedding-based retrieval and relevance scoring, improving both recall and precision while preserving the original document semantics. Overall, this design ensures that retrieval at inference time yields coherent and contextually complete knowledge snippets, effectively guiding downstream code generation and improving the robustness of LLM-based CAD reasoning.

5. Experiments and Comparison

This section details the implementation of CADDesigner and the comprehensive experimental evaluation conducted to assess its performance in CAD modeling code generation.

5.1. Implementation Details

We evaluate our CAD modeling code generation algorithm on a workstation with an 8-core Intel CPU under Python 3.12. The system is implemented on a customized agent framework that leverages retrieval-augmented generation (RAG) via RAGFlow. For embedding-based retrieval, we utilize the embedding model bge-large-zh-v1.5 to encode knowledge base documents. The retrieval employs a hybrid similarity scoring method, where vector similarity and keyword-based similarity are combined with weights of 0.6 and 0.4, respectively. The top 3 most relevant documents (top-k=3) are retrieved for downstream generation. For LLM services, the primary agent and the code generation tool (T_2) employ Claude-4-Sonnet, while requirement refinement (T_1) and both visual feedback sub-tools (visual question generation and visual feedback generation) are based on Gemini-2.5-Flash. Inference uses a temperature of 0.7 and nucleus sampling (top-p) of 0.9 to balance diversity and coherence. All experiments are conducted without the user feedback loops (R and Sat), so the reported results do not include online case accumulation driven by user approval.

5.2. Dataset

Our experiments are conducted on the large-scale DeepCAD dataset [1], which contains approximately 178K parametric CAD models represented in a sketch-and-extrude format. Following SkexGen [10], we first remove duplicate models to reduce redundancy. From the training set, we randomly select $1K$ models to initialize the RAG knowledge base. In addition, we develop a conversion script to transform the parametric command sequences of each model into ECIP code snippets, which are included in the RAG knowledge base as `Case`. For evaluation, we construct two test subsets from the de-duplicated DeepCAD test set. The first subset contains 200 models, selected using uniform stratified sampling based on the number of commands in the sequence (1-10, 11-20, 21-30, 31-40, 41+) to ensure coverage across different complexity levels. The second subset contains $1K$ models randomly sampled from the same de-duplicated test set and is used for comparison with prior text-to-CAD methods. For each sampled model, a single representative image is rendered from a fixed camera viewpoint to provide visual input for the experiments, while the textual inputs are derived from the abstract-level descriptions provided in the DeepCAD dataset.

5.3. Evaluation Metrics

To assess the geometric fidelity of the generated CAD models, we adopt several standard shape similarity metrics. Let $\mathcal{S}$ denote the reference models and $\mathcal{G}$ represent the generated models, with $\mathcal{X}$ and $\mathcal{Y}$ representing their corresponding point clouds. These metrics evaluate the agreement between generated and ground-truth geometries from volumetric and point-wise perspectives.

Intersection over Union (IoU) is utilized to measure the similarity between generated models and the ground truth.

$$\mathrm{IoU}(\mathcal{G}, \mathcal{S}) = \frac{\mathcal{G} \cap \mathcal{S}}{\mathcal{G} \cup \mathcal{S}}, \tag{10}$$

where $\mathcal{G} \cap \mathcal{S}$ denotes the overlapping volume between the reference and generated models, and $\mathcal{G} \cup \mathcal{S}$ represents their combined volumetric union. A value of 1 indicates perfect alignment, while 0 signifies no overlap.

Chamfer Distance (CD) calculates point-wise proximity between point clouds sampled from $\mathcal{X}$ and $\mathcal{Y}$:

$$\begin{aligned}\mathrm{CD}(\mathcal{X}, \mathcal{Y}) = &\frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \min_{y \in \mathcal{Y}} \|x - y\|_2^2 \\ &+ \frac{1}{|\mathcal{Y}|} \sum_{y \in \mathcal{Y}} \min_{x \in \mathcal{X}} \|y - x\|_2^2.\end{aligned} \tag{11}$$

Hausdorff Distance (HD) is a metric for quantifying the similarity between two point sets, primarily used for evaluating the resemblance of object contours.

$$\begin{aligned}\mathrm{HD}(\mathcal{X}, \mathcal{Y}) = \max \Big\{ &\sup_{x \in \mathcal{X}} \inf_{y \in \mathcal{Y}} d(x, y), \\ &\sup_{y \in \mathcal{Y}} \inf_{x \in \mathcal{X}} d(x, y) \Big\},\end{aligned} \tag{12}$$

where $d(x, y)$ is the Euclidean distance between points x and y, sup (supremum) and inf (infimum) represent the least upper bound and greatest lower bound, respectively.

To ensure fair and reproducible comparison, all geometric metrics are evaluated under standardized **metric computation protocols**, including normalization, alignment, voxelization, and point cloud sampling, which are applied consistently across all methods and baselines:

- **Normalization and Alignment:** All meshes and sampled point clouds are first normalized to fit within a unit cube $[-0.5, 0.5]^3$. For pairwise comparisons, rigid alignment between predicted and ground-truth shapes is performed using the Iterative Closest Point (ICP) algorithm to eliminate differences due to translation and rotation.
- **IoU Voxelization:** For IoU computation, both meshes are voxelized using a fixed voxel size of 0.02. The voxel grid is defined over the union of the bounding boxes of both models, with an additional margin of two voxels to ensure full coverage.
- **Point Cloud Sampling:** For Chamfer Distance (CD) and Hausdorff Distance (HD), 2048 points are uniformly sampled from the surface of each mesh. This sampling density is kept consistent across all methods.

In addition, we introduce several process-oriented metrics to analyze modeling stability, reliability, and efficiency under ablation settings: **Pass@1** denotes the percentage of cases where valid code is generated on the first attempt, reflecting generation accuracy. **AVG Re** (Average Retry) captures the mean number of retries required during the iterative correction loop, indicating convergence efficiency. **SUC** denotes the proportion of cases where a syntactically valid CAD model is successfully generated, i.e., models that can be executed and rendered without errors. Note that a syntactically valid model is not necessarily semantically aligned with the input description. **Tokens** measures the average number of tokens consumed to generate a single CAD model. **Latency** measures the average time required to generate a single CAD model.

5.4. Comprehensive Ablation Study

To comprehensively verify the design choices and the effectiveness of different components in our CADDesigner agent, we conduct four groups of ablation studies using the 200-model test subset described in Section 5.2: one focusing on the ECIP design, another evaluating the impact of CAD tools and input modalities, a third analyzing the effect of different LLM backends, and a fourth investigating how model complexity influences token usage and generation latency. Unless otherwise specified, all experiments are conducted with text-only input.

5.4.1. Effect of ECIP Design Components

To illustrate the benefits of the ECIP design choices, such as the error handling mechanism that is friendly to LLMs and the overall architecture of the ECIP system itself, we evaluate three ECIP variants (full ECIP, ECIP without the detailed error system, and ECIP without the return type annotations).

The results in Table 2 show that the full ECIP configuration achieves the best overall performance among all variants. When the error handling mechanism is removed (ECIP w/o Err), the Pass@1 decreases slightly from 0.45 to 0.41, while the AVG Re increases from 1.86 to 2.62, and SUC drops to 81.5%. This suggests that although the agent can still generate correct code initially, the lack of detailed feedback slows down convergence during iterative correction and reduces the overall success rate. In contrast, removing type annotations (ECIP w/o Type) causes Pass@1 to drop significantly from 0.45 to 0.32, indicating that the initial code generation is more prone to semantic errors without explicit type guidance. The AVG Re (2.30) is higher than full ECIP, but still lower than ECIP without error handling, and SUC also drops to 90.5%. This shows that type annotations help encode critical modeling intent into the prompt, yet some type errors can still be corrected through retries. Overall, we observe a clear division of roles: error handling helps the agent quickly recover from mistakes during iterations, and type annotations improve first-attempt accuracy by reducing semantic ambiguity.

Table 2
Ablation study of ECIP design components on 200 test models.

	Pass@1↑	AVG Re↓	SUC↑
ECIP	**0.45**	**1.86**	**100.0%**
ECIP w/o Err	0.41	2.62	81.5%
ECIP w/o Type	0.32	2.30	90.5%

Table 3
Ablation study of CADDesigner components and input modalities on 200 test models.

	IoU↑	CD↓	HD↓
A (w/o `Anno`)	–	–	–
B (w/o `Case`)	0.2650	0.1350	0.4690
C (w/o T_1)	0.2726	0.1251	0.5595
D (w/o T_4)	0.2522	0.1192	0.4793
E (Text only)	0.3041	0.1236	0.4154
F (Image only)	0.3518	0.1167	0.4262
G (Text + Image)	**0.3893**	**0.0693**	**0.2553**

5.4.2. Effect of CAD Tools and Input Modalities

To further analyze the individual contributions of the key components in CADDesigner, we evaluate seven variants, labeled A to G, which respectively correspond to: A) removal of function annotations `Anno` from the knowledge base, B) removal of basic model cases `Case`, C) disabling the requirement refinement tool T_1, D) disabling the visual feedback tool T_4, E) the full CADDesigner pipeline with text-only input, F) full pipeline with image-only input, and G) full pipeline with combined text and image input. Except for variants F and G, all others use text input exclusively.

Table 3 summarizes the quantitative results across multiple evaluation metrics. In Model A, removing semantic API annotations (`Anno`) prevents the agent from reliably identifying available operations and their usage, leading to failure to generate valid models. This indicates that structured function-level annotations play a critical role in grounding agent-based CAD code generation and guiding convergence toward correct solutions.

Models B, C, and D disable key system components–case examples (`Case`), requirement refinement (T_1), and visual feedback (T_4), respectively. Compared to the text-only variant

Table 4
Ablation study of different LLM backends on 200 test models for ECIP code generation.

LLM Backend	IoU↑	CD↓	HD↓
Claude-4-Sonnet	**0.3041**	**0.1236**	**0.4154**
Gemini-2.5-Pro	0.2951	0.1571	0.4902
GPT-5.2-Codex	0.2914	0.1604	0.5021

(Model E), all three models show degraded performance in terms of IoU (0.2650, 0.2726, 0.2522 vs. 0.3041), suggesting that each component contributes positively to robust and accurate model generation. Specifically, removing structured case examples and requirement refinement increases semantic ambiguity during early-stage generation, while disabling visual feedback leads to the most significant drop in IoU, highlighting its critical role in iterative refinement and overall structural correctness.

Switching the input modality from text-only (Model E) to image-only (Model F) yields a notable performance improvement, suggesting that abstract textual descriptions alone are often insufficient to fully specify geometric details, whereas images provide richer spatial cues for CAD modeling. Model G, which combines both text and image as input, achieves the best results across all metrics, illustrating that textual information can effectively complement visual context by resolving ambiguities difficult to infer from images alone.

Overall, these results indicate that structured API knowledge, coordinated tool usage, and multimodal input jointly contribute to high-fidelity and robust CAD model generation.

5.4.3. Effect of LLM Backend

To investigate the influence of different LLM backends on CAD code generation, we conduct an ablation study by replacing the LLM backend used in the code generation tool (T_2) of CADDesigner with three representative models: Claude-4-Sonnet, Gemini-2.5-Pro, and GPT-5.2-Codex. Each backend is evaluated under the ECIP code paradigm on 200 test models.

The quantitative results are reported in Table 4. We observe that CADDesigner achieves consistent performance across the three LLM backends evaluated. Claude-4-Sonnet attains the highest geometric fidelity, reflected by slightly higher IoU and lower CD and HD values. Gemini-2.5-Pro and GPT-5.2-Codex show comparable results, with small differences in CD and HD relative to Claude-4-Sonnet. These observations suggest that, among LLMs of similar capability, CADDesigner produces stable and reliable CAD code. The ECIP representation explicitly manages intermediate modeling states, and the knowledge-constrained generation framework provides structured guidance and reference cases, helping the code generation tool (T_2) produce scripts that are structurally correct and semantically meaningful. Overall, these results suggest that CADDesigner maintains broadly consistent CAD code generation across these backends, with moderate variations attributable to the underlying LLM.

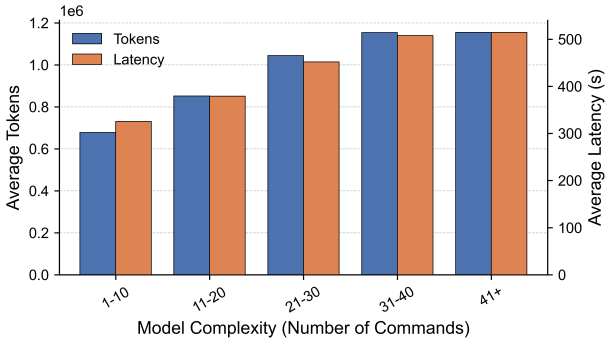

Figure 4: Token usage and generation latency as a function of model complexity (number of commands). Models are grouped into bins of 10 commands each.

5.4.4. Effect of Model Complexity on Inference Cost

We evaluate the impact of model complexity on inference cost in CADDesigner. Specifically, we focus on two main metrics: Tokens and Latency. For memory consumption, CADDesigner primarily relies on external LLM APIs, resulting in negligible GPU memory overhead, while the main memory usage comes from RAG components (e.g., RAGFlow), which remain relatively stable and do not scale significantly with model complexity. Models are grouped into bins with an interval of 10 commands. For each complexity bin, we report the average Tokens and average Latency over all models within the bin. As shown in Figure 4, both Tokens and Latency increase as model complexity grows. This is expected, since more complex models require the agent to generate longer CAD operation sequences and perform more reasoning steps during code generation. However, the increase is relatively moderate rather than steep. From 1–10 to 41 or more commands, token usage and latency grow steadily without abrupt escalation, indicating that the system does not incur excessive overhead as complexity increases. This behavior can be attributed to the design of CADDesigner. The explicit context representation in ECIP and the structured code generation process in T_2 help maintain stable reasoning and avoid redundant or excessively long intermediate steps. As a result, the inference cost scales in a controlled manner with model complexity.

We further examine the average contribution of each component in the pipeline across all 200 test models. Figure 5 shows the average proportion of Tokens and Latency for the primary agent and the CAD tools. The code generation tool T_2 accounts for the largest share in both Tokens and Latency. The Requirement Refinement Tool T_1 and the Visual Feedback Tool T_4 each contribute relatively small portions, while the Command Execution Tool T_3 contributes modestly to latency without consuming tokens. The primary agent itself has only a minor contribution in both metrics. These observations indicate that T_2 is the main driver of both token and latency costs, whereas other components have limited impact on overall inference cost.

5.5. Comparison Methods

We compare CADDesigner from two complementary perspectives: (1) different CAD representation paradigms

Table 5
Comparison of different CAD representation paradigms on 200 test models.

Representation	IoU↑	CD↓	HD↓	Pass@1↑	AVG Re↓	SUC↑	Tokens (M)↓	Latency (s)↓
ECIP	**0.3041**	**0.1236**	**0.4154**	0.46	**1.86**	**100.0%**	**0.98**	436
CadQuery	0.2827	0.1818	0.5186	0.50	2.01	87.5%	1.40	541
build123d	0.2617	0.1570	0.5126	**0.59**	1.91	96.0%	1.35	**363**

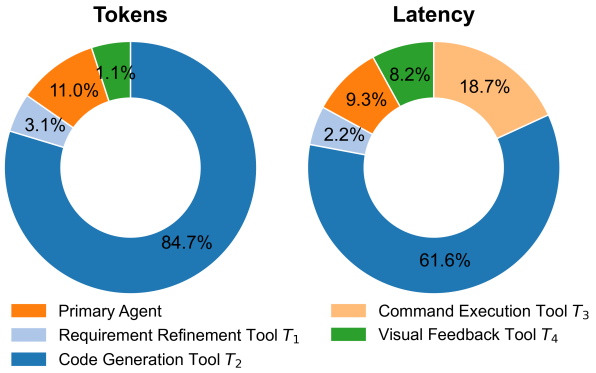

Figure 5: Average inference cost breakdown across the CAD-Designer pipeline. Tokens (left) and Latency (right) for each component.

used for code generation, and (2) existing text-to-CAD generation methods.

5.5.1. Comparison of CAD Representation Paradigms

We first evaluate the impact of different CAD representation paradigms on code generation performance on 200 test models. Specifically, we compare our proposed ECIP paradigm with two widely used alternatives: CadQuery and build123d. To ensure a fair comparison, we integrate CadQuery and build123d into the same agent framework as CADDesigner by replacing the ECIP-based code generation component, while keeping all other components unchanged as described in Section 5.1. We further equip both CadQuery and build123d with RAG modules constructed from their official documentation. All methods use text-only input for evaluation. Here, ECIP, CadQuery, and build123d should be understood as different code representation styles or API interfaces used for generation, rather than fundamentally different geometric kernels.

The quantitative results presented in Table 5 illustrate clear differences among the evaluated CAD code representations. ECIP achieves the best IoU and SUC, as well as the lowest AVG Re, indicating stronger geometric fidelity and more reliable iterative convergence. Since ECIP is implemented as a lightweight representation layer on top of CadQuery, these differences should be understood primarily at the level of code representation and LLM usability rather than the underlying geometric engine. In particular, although CadQuery can also support coding patterns that partially overlap with ECIP, its commonly used fluent interface does not explicitly standardize intermediate modeling context for language models. By contrast, ECIP consistently organizes each operation into explicit object-and-parameter transformations, which makes dependency tracking, code generation, and iterative refinement more reliable for LLM-based agents. Build123d attains the highest Pass@1 and shortest latency, indicating that its compact syntax can often be quickly converted into executable code. However, its lower IoU and higher CD and HD suggest that executable code does not necessarily imply geometric accuracy, which may be due to its reliance on overloaded symbolic operators, making modeling semantics harder for LLMs to interpret than explicit function calls. Overall, the results highlight trade-offs among representation styles for LLM-oriented CAD coding: APIs that simplify LLM code generation can improve early success and reduce execution time, but may compromise geometric accuracy.

To further assess robustness and distribution-level differences among ECIP, CadQuery, and build123d, we analyze the distributions of IoU, CD, and HD across the 200 test models. As shown in Figure 6, ECIP exhibits higher median IoU and lower CD/HD values with tighter distributions, indicating more consistent geometric fidelity. In comparison, CadQuery and build123d show wider variability and higher CD/HD, highlighting less precise and less reliable CAD model generation. These results suggest that ECIP demonstrates more accurate and consistent performance across the evaluated models.

5.5.2. Comparison with Existing Methods

We further compare our method with two learning-based text-to-CAD generation methods: Text2CAD [4], cadrille [28] and one agent-based method: CADCodeVerify [25].

- **Text2CAD** proposes an end-to-end transformer-based autoregressive network to generate parametric CAD models from input texts.
- **cadrille** proposes a two-stage CAD reconstruction pipeline: supervised fine-tuning (SFT) on large-scale procedurally generated data, followed by reinforcement learning fine-tuning using online feedback.
- **CADCodeVerify** proposes an agent that utilizes validation questions and visual feedback for design verification.

For evaluation, we use the $1K$-model subset described in Section 5.2, randomly sampled from the de-duplicated DeepCAD test set. For each instance, we use the abstract text description as input. We then perform inference using their released weights and our method to generate results

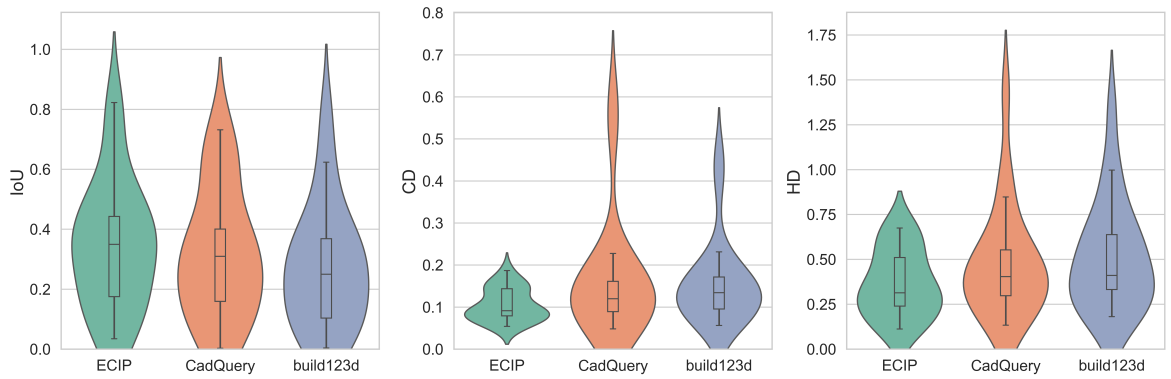

Figure 6: Violin plots comparing the distributions of IoU, CD, and HD across ECIP, CadQuery, and build123d on 200 test models. ECIP achieves higher geometric fidelity and more consistent performance compared to the other paradigms.

Table 6
Performance comparison of different Text-to-CAD methods under abstract text inputs on $1K$ test models.

Method	IoU↑	CD↓	HD↓	SUC↑
Text2CAD	0.1831	0.1475	0.5680	96.6%
cadrille	0.0274	0.2162	0.5817	98.2%
CADCodeVerify	0.2348	0.2329	0.4892	86.1%
CADDesigner	**0.2769**	**0.1097**	**0.4347**	**100.0%**

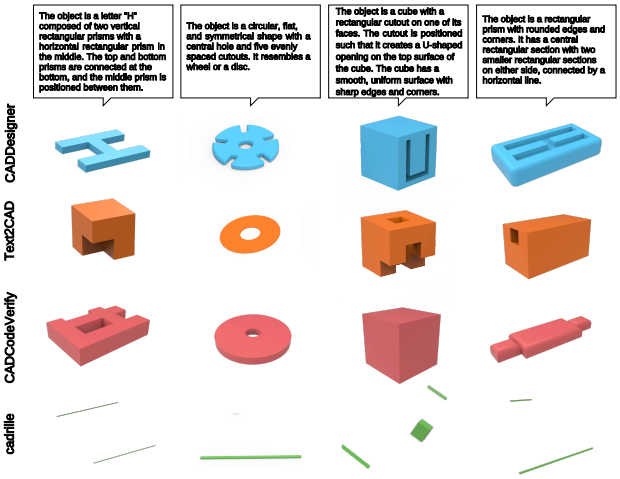

Figure 7: Comparison of generation results across Text2CAD, CADCodeVerify, cadrille, and our method. CADDesigner achieves the best input alignment. Text2CAD and CADCodeVerify show moderate performance, while cadrille generates syntactically valid but semantically incorrect outputs due to poor generalization from expert-level training to abstract inputs.

for metric calculation. As presented in Table 6 and Figure 7, the experimental results show that CADDesigner achieves the best performance. CADDesigner demonstrates the best prompt-result alignment, followed by CADCodeVerify and Text2CAD. While cadrille achieves a high SUC score, indicating it can produce syntactically valid CAD models in most cases, the visualized results reveal these outputs are typically meaningless with respect to the input instructions (see Figure 7). This failure is due to cadrille's poor generalization from expert-level to abstract instructions: it generates

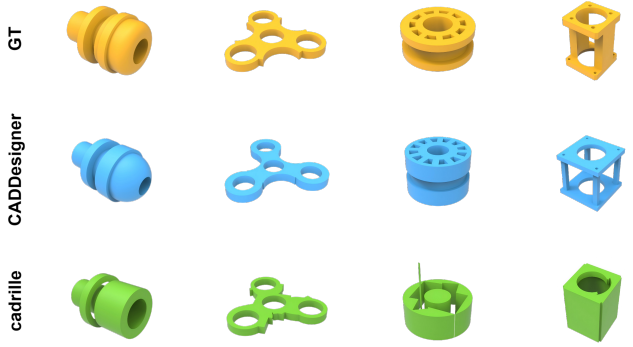

Figure 8: Comparison of CADDesigner and cadrille on image-based inputs. CADDesigner (blue) generates CAD models that more closely match the input images on these representative conceptual cases, benefiting from support for operations such as revolve and pattern-based constructions. In contrast, cadrille (green), relying on low-level extrusion, often produces geometrically less faithful results.

structurally valid but semantically irrelevant models. In contrast, the other methods generate outputs that are both syntactically and semantically valid to varying degrees, with our method being the most consistent.

Additionally, we compare our method with cadrille on image-based inputs to further illustrate their respective modeling capabilities. As shown in Figure 8, CADDesigner consistently generates CAD models that closely match the input images, benefiting from a more expressive CAD operation space. In particular, several models require rotationally symmetric structures that are naturally constructed via revolve operations, which are explicitly supported by CADDesigner but not by cadrille, leading to missing or distorted geometries in the latter. Moreover, models with rich geometric details and repetitive structures are more effectively specified using high-level CAD operations such as arrays or patterned constructs. CADDesigner can leverage these higher-level APIs to compactly and precisely generate such structures, whereas cadrille relies primarily on low-level extrusion-based operations, often resulting in geometrically less faithful outputs.

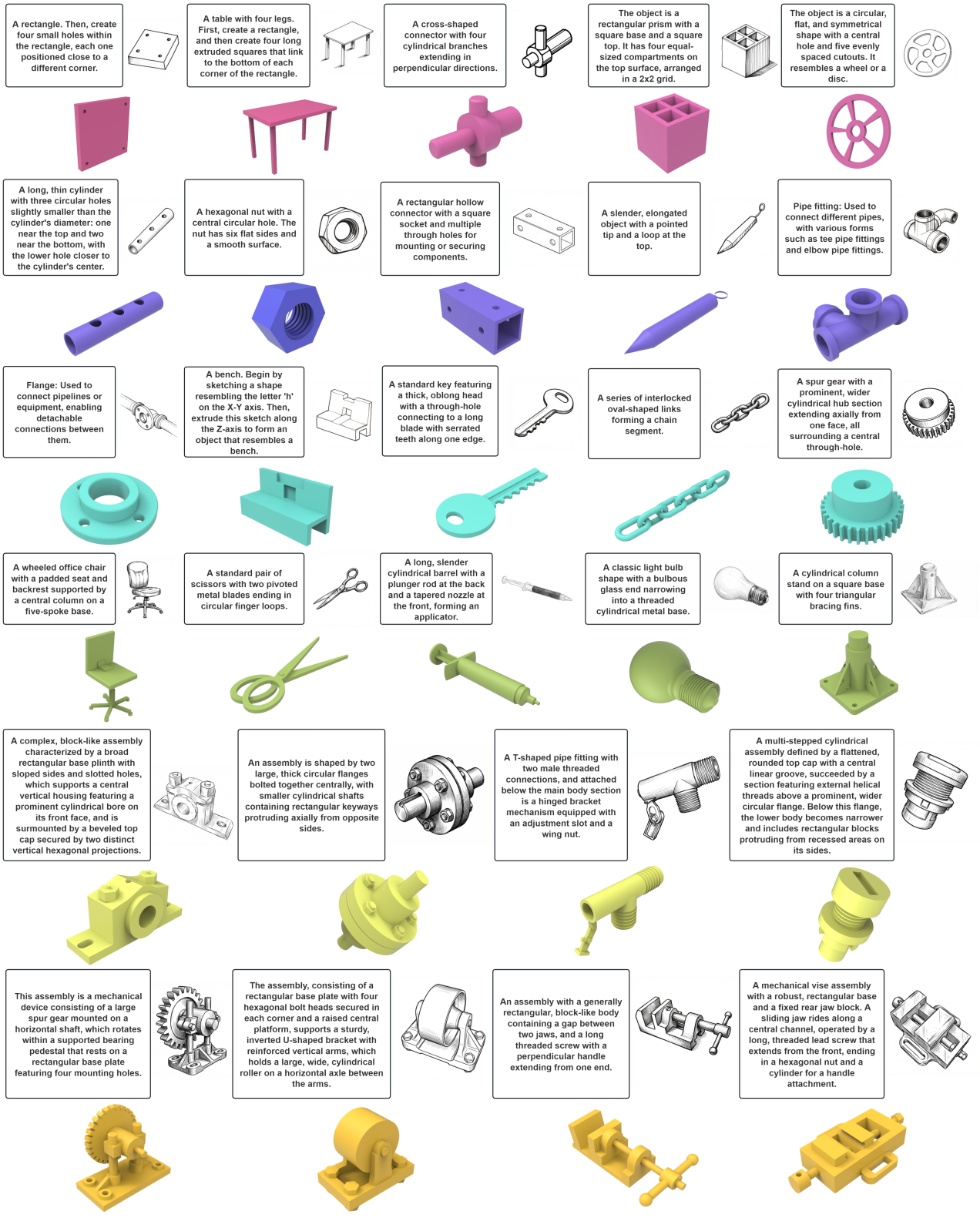

Figure 9: Visual results with sketch-text input.

These results highlight that beyond syntactic validity, the availability of expressive and semantically aligned CAD operations is crucial for faithful image-based CAD model generation.

5.6. Visual Results Presentation

We additionally provide qualitative results generated from sketch–text inputs, with representative examples illustrated in Figure 9. The content presented here is derived from the DeepCAD dataset, the CADCodeVerify dataset, and datasets covering a wide range of geometries, including standard industrial components (e.g., bolts, flanges, and pipe fittings), as well as more complex models and assemblies.

5.7. Limitations

Although the latency of CADDesigner scales in a relatively controlled manner with increasing model complexity, its end-to-end response time remains non-negligible in interactive usage scenarios, particularly for complex model generation and multi-round iterative refinement. Our analysis as shown in Figure 5 reveals that the code generation module (T_2) is the primary contributor to the overall system latency. Therefore, improving inference efficiency remains a critical challenge.

Furthermore, the current framework does not yet sufficiently incorporate the domain-specific knowledge required in industrial design practice, such as manufacturing constraints, tolerance requirements, assembly semantics, and standardized engineering conventions. Therefore, the capability for complex model generation and validation in specific industrial domains still requires further improvement.

6. Conclusion and Future Work

We propose CADDesigner, a novel framework for CAD modeling code generation, combined with an explicit context imperative paradigm to produce high-quality CAD modeling scripts. Experimental results show that CADDesigner achieves competitive performance and outperforms the evaluated baselines in conceptual CAD design tasks. CADDesigner is particularly well-suited for rapid prototyping and early-stage conceptual CAD modeling, where user intent is often abstract and iterative refinement plays a critical role.

In future work, we plan to extend CADDesigner by incorporating learning-based methods that accept point clouds and B-rep data as input, allowing the system to learn geometric and topological constraints through data-driven training. Furthermore, we aim to further optimize the inference efficiency of the framework and introduce domain-specific industrial cases and functional constraints, thereby improving the generation capability and efficiency of our method for complex models in industrial scenarios.

References

[1] Rundi Wu, Chang Xiao, and Changxi Zheng. DeepCAD: A deep generative network for computer-aided design models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 6772–6782, 2021.

[2] Xueyang Li, Yunzhong Lou, Yu Song, and Xiangdong Zhou. Mamba-CAD: State space model for 3D computer-aided design generative modeling. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 5013–5021, 2025.

[3] Mohammad Sadil Khan, Elona Dupont, Sk Aziz Ali, Kseniya Cherenkova, Anis Kacem, and Djamila Aouada. CAD-SIGNet: CAD language inference from point clouds using layer-wise sketch instance guided attention. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4713–4722, 2024.

[4] Mohammad Sadil Khan, Sankalp Sinha, Talha Uddin Sheikh, Didier Stricker, Sk Aziz Ali, and Muhammad Zeshan Afzal. Text2CAD: Generating sequential CAD designs from beginner-to-expert level text prompts. In *Advances in Neural Information Processing Systems*, volume 37, pages 7552–7579, 2024.

[5] Jiahao Li, Weijian Ma, Xueyang Li, Yunzhong Lou, Guichun Zhou, and Xiangdong Zhou. CAD-Llama: leveraging large language models for computer-aided design parametric 3D model generation. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 18563–18573, 2025.

[6] Siyu Wang, Cailian Chen, Xinyi Le, Qimin Xu, Lei Xu, Yanzhou Zhang, and Jie Yang. CAD-GPT: Synthesising CAD construction sequence with spatial reasoning-enhanced multimodal LLMs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 7880–7888, 2025.

[7] Danila Rukhovich, Elona Dupont, Dimitrios Mallis, Kseniya Cherenkova, Anis Kacem, and Djamila Aouada. CAD-Recode: Reverse engineering CAD code from point clouds. *arXiv preprint arXiv:2412.14042*, 2024.

[8] CadQuery contributors. CadQuery, February 2026.

[9] Karl D. D. Willis, Yewen Pu, Jieliang Luo, Hang Chu, Tao Du, Joseph G. Lambourne, Armando Solar-Lezama, and Wojciech Matusik. Fusion 360 gallery: A dataset and environment for programmatic CAD construction from human design sequences. *ACM Transactions on Graphics (TOG)*, 40(4), 2021.

[10] Xiang Xu, Karl DD Willis, Joseph G Lambourne, Chin-Yi Cheng, Pradeep Kumar Jayaraman, and Yasutaka Furukawa. SkexGen: Autoregressive generation of CAD construction sequences with disentangled codebooks. In *International Conference on Machine Learning*, pages 24698–24724, 2022.

[11] Aijia Zhang, Weiqiang Jia, Qiang Zou, Yixiong Feng, Xiaoxiang Wei, and Ye Zhang. Diffusion-CAD: Controllable diffusion model for generating computer-aided design models. *IEEE Transactions on Visualization and Computer Graphics*, 2025.

[12] Haoxiang Guo, Shilin Liu, Hao Pan, Yang Liu, Xin Tong, and Baining Guo. ComplexGen: CAD reconstruction by B-rep chain complex generation. *ACM Transactions on Graphics (TOG)*, 41(4):1–18, 2022.

[13] Weijian Ma, Shuaiqi Chen, Yunzhong Lou, Xueyang Li, and Xiangdong Zhou. Draw step by step: reconstructing CAD construction sequences from point clouds via multimodal diffusion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 27154–27163, 2024.

[14] Yuan Li, Cheng Lin, Yuan Liu, Xiaoxiao Long, Chenxu Zhang, Ningna Wang, Xin Li, Wenping Wang, and Xiaohu Guo. CADDreamer: CAD object generation from single-view images. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 21448–21457, 2025.

[15] Cheng Chen, Jiacheng Wei, Tianrun Chen, Chi Zhang, Xiaofeng Yang, Shangzhan Zhang, Bingchen Yang, Chuan-Sheng Foo, Guosheng Lin, Qixing Huang, and Fayao Liu. CADCrafter: Generating computer-aided design models from unconstrained images. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 11073–11082, 2025.

[16] Sifan Wu, Amir Khasahmadi, Mor Katz, Pradeep Kumar Jayaraman, Yewen Pu, Karl Willis, and Bang Liu. CAD-LLM: Large language model for cad generation. In *Proceedings of the neural information processing systems conference*, 2023.

[17] Sifan Wu, Amir Hosein Khasahmadi, Mor Katz, Pradeep Kumar Jayaraman, Yewen Pu, Karl Willis, and Bang Liu. CadVLM: Bridging

language and vision in the generation of parametric CAD sketches. In *European Conference on Computer Vision*, pages 368–384, 2024.
[18] Jingwei Xu, Zibo Zhao, Chenyu Wang, Wen Liu, Yi Ma, and Shenghua Gao. CAD-MLLM: Unifying multimodality-conditioned CAD generation with MLLM. *arXiv preprint arXiv:2411.04954*, 2024.
[19] Qwen Team. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024.
[20] Haoyang Xie and Feng Ju. Text-to-CadQuery: A New Paradigm for CAD generation with scalable large model capabilities. *arXiv preprint arXiv:2505.06507*, 2025.
[21] Dimitrios Mallis, Ahmet Serdar Karadeniz, Sebastian Cavada, Danila Rukhovich, Niki Foteinopoulou, Kseniya Cherenkova, Anis Kacem, and Djamila Aouada. CAD-Assistant: Tool-augmented VLLMs as generic CAD task solvers. *arXiv preprint arXiv:2412.13810*, 2024.
[22] Juergen Riegel, Werner Mayer, and Yorik van Havre. FreeCAD: Open-source parametric 3D CAD modeler, 2024.
[23] Xueyang Li, Jiahao Li, Yu Song, Yunzhong Lou, and Xiangdong Zhou. Seek-CAD: A self-refined generative modeling for 3D parametric CAD using local inference via DeepSeek. *arXiv preprint arXiv:2505.17702*, 2025.
[24] Zeqing Yuan, Haoxuan Lan, Qiang Zou, and Junbo Zhao. 3D-PreMise: Can large language models generate 3D shapes with sharp features and parametric control? *arXiv preprint arXiv:2401.06437*, 2024.
[25] Kamel Alrashedy, Pradyumna Tambwekar, Zulfiqar Zaidi, Megan Langwasser, Wei Xu, and Matthew Gombolay. Generating CAD code with vision-language models for 3d designs. *arXiv preprint arXiv:2410.05340*, 2024.
[26] Roger Maitland, jdegenstein, Bernhard, Ethan Rooke, JR Mobley, snoyer, Jojain, Andreas Felix Häberle, Ruud, Ami Fischman, Jason S. McMullan, Roman Dvořák, simon klemenc, BogdanTheGeek, Spectre5, Dalibor Frívaldský, Daniele D'Orazio, George, hoijui, OpenVMP, Yeicor, Alexander Steppke, mayhem 64, luzpaz, nobkd, Victor Poughon, slobberingant, Arno Bosch, Barnaby Walters, and Matti Eiden. gumyr/build123d: v0.9.1, feb 2025.
[27] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023.
[28] Maksim Kolodiazhnyi, Denis Tarasov, Dmitrii Zhemchuzhnikov, Alexander Nikulin, Ilya Zisman, Anna Vorontsova, Anton Konushin, Vladislav Kurenkov, and Danila Rukhovich. cadrille: Multi-modal CAD reconstruction with online reinforcement learning. *arXiv preprint arXiv:2505.22914*, 2025.

A. Generation Instance

We give a complete end-to-end example to illustrate how CADDesigner transforms a user request into an executable CAD model. Given the input *"Create a model of a mechanical component featuring a large circular flange base and a central hollow shaft boss"*, the system first performs requirement refinement to expand the coarse description into a detailed design specification. This process includes identifying key geometric parameters, structural relationships, and a feasible modeling procedure, as shown in Figure A.1. The API documentation in the knowledge base provides structured information such as function signatures, parameter types, return values, and usage examples, which help align natural language intent with executable CAD operations, as shown in Listing A.1. After the user's requirements are clearly defined, the intelligent agent retrieves the knowledge base and generates code-oriented guidance, as shown in Figure A.2. Based on the refined design specification and retrieved knowledge, we utilize the prompt illustrated in Figure A.3 to invoke the code generation tool, thereby producing executable ECIP-based CAD code. The generated script explicitly constructs geometric primitives, applies spatial transformations, and performs boolean operations to assemble the final model. After execution, the system produces the CAD model together with structured geometry metadata, including key–value fields and nested attributes that capture semantic properties, geometric parameters, and derived physical quantities (see Listing A.2). The generated CAD model is then rendered into multiple views and evaluated through visual feedback. The system formulates requirement-aware visual questions and verifies whether the generated geometry satisfies the intended design. If inconsistencies are detected, corrective suggestions are produced to guide further refinement; otherwise, the model is accepted as the final output. Figure A.4 illustrates the prompt structure used for visual feedback generation, and Listing A.3 presents the complete, validated implementation for generating the mechanical component. Finally, the ECIP script is mapped to a FreeCAD macro through a conversion script as shown in Figure A.5, thereby supporting the editing of modeling sequences within the CAD environment.

This example demonstrates the full workflow of CAD-Designer, including requirement refinement, knowledge-grounded code generation, and iterative validation, forming a complete pipeline from user input to validated CAD output.

Basic Expansion

Requirements:
1. Build a mechanical component consisting of a circular flange base, a main vertical shaft, and a perpendicular cylindrical boss near the top.
2. The flange base is a circular disk with specified radius and thickness.
3. The main shaft is a vertical cylinder positioned concentrically on top of the flange base.
4. The perpendicular boss is a cylindrical structure located near the upper region of the main shaft and extends horizontally through the shaft.
5. A central vertical bore passes through the base and the main shaft.
6. A smaller horizontal hole passes through the perpendicular boss.
7. Finally, combine the flange base, the main shaft, and the boss into a complete solid structure, and subtract the bore and boss hole to obtain the final model.

Parameter Description Table:

Parameter	Type	Default Value	Calculation Logic
`base_radius`	float	30.0	Radius of the flanged base
`base_height`	float	5.0	Thickness of the flanged base
`shaft_radius`	float	10.0	External radius of the main vertical shaft
`shaft_height`	float	60.0	Height of the shaft above the base
`bore_radius`	float	6.0	Radius of the central vertical bore
`boss_radius`	float	8.0	Radius of the perpendicular boss cylinder
`boss_length`	float	20.0	Total length of the boss
`boss_z_offset`	float	50.0	Height of the boss axis center from the base bottom
`boss_hole_radius`	float	4.0	Radius of the hole through the boss

Modeling Process:
Steps:
1. **Create the Base Flange**
   - **Purpose:** Construct the circular flanged base of the component.
   - **API:** `make_cylinder_rsolid`

...
7. **Subtract the Hole Features**
   - **Purpose:** Subtract the central bore and the boss hole from the merged solid to form the final geometry.
   - **API:** `cut_rsolidlist`

Figure A.1: Detailed requirements analysis without API details.

Detailed Requirements (D_{detail})

Refined Requirements:
1. Build a mechanical component consisting of a circular flange base, a main vertical shaft, and a perpendicular cylindrical boss near the top.
2. The flange base is a circular disk with specified radius and thickness.
3. The main shaft is a vertical cylinder positioned concentrically on top of the flange base.
4. The perpendicular boss is a cylindrical structure located near the upper region of the main shaft and extends horizontally through the shaft.
5. A central vertical bore passes through the base and the main shaft.
6. A smaller horizontal hole passes through the perpendicular boss.
7. Finally, combine the flange base, the main shaft, and the boss into a complete solid structure, and subtract the bore and boss hole to obtain the final model.

Parameter Description Table:

Parameter	Type	Default Value	Calculation Logic
`base_radius`	float	30.0	Radius of the flanged base
`base_height`	float	5.0	Thickness of the flanged base
`shaft_radius`	float	10.0	External radius of the main vertical shaft
`shaft_height`	float	60.0	Height of the shaft above the base
`bore_radius`	float	6.0	Radius of the central vertical bore
`boss_radius`	float	8.0	Radius of the perpendicular boss cylinder
`boss_length`	float	20.0	Total length of the boss
`boss_z_offset`	float	50.0	Height of the boss axis center from the base bottom
`boss_hole_radius`	float	4.0	Radius of the hole through the boss

Modeling Process:
Steps:
1. **Create the Base Flange**
   - **Purpose:** Construct the circular flanged base.
   - **API:** `make_cylinder_rsolid`
   - **Code Snippet:**
   ```python
   base = make_cylinder_rsolid(radius=base_radius,
                               height=base_height,
                               bottom_center=(0, 0, 0)
   )
   ```
...
7. **Subtract the Hole Features**
   - **Purpose:** Subtract the bore and boss hole from the combined solid.
   - **API:** `cut_rsolidlist`
   - **Code Snippet:**
   ```python
   final_model = cut_rsolidlist(combined, [bore, boss_hole])
   ```

Figure A.2: Detailed requirements analysis with API details and code snippets based on querying the knowledge base.

```
# Background Info:
The SimpleCAD API provides a complete set of geometric modeling classes, from basic points, lines, and surfaces to complex solids and compounds.
Each class has rich functions and a flexible label management system.
### Basic Classes
#### [CoordinateSystem - Coordinate System]
#### [SimpleWorkplane - Workplane]
The workplane context manager is used with "with" to provide a local coordinate system environment and supports nested use.
### Geometry Classes
##### [Vertex - Vertex]
... [ some background information about primitive type and methods. ]
## Coordinate System
SimpleCAD uses a unified coordinate system:
...  [ coordinate system information here ]

## Design Principles
### Consistency
All classes follow the same design pattern and naming convention (V+N+returntype) to provide a consistent user experience.

## Usage Guide
1. **Create Geometries**: Use `make_*` functions to create basic geometries
4. **Combination Operations**: Use boolean operations, transformations, etc., to create complex geometries
5. **Query and Filter**: Query required geometries based on labels and metadata
### Best Practices
1. **Label Naming**: Use set_tag to set labels for geometric elements, such as `category.subcategory.detail`
2. **Coordinate System Management**: Reasonably use workplanes to simplify the creation of complex geometries
3. **Check Necessary Translations**: Pay attention to where the final result of the advanced Solid construction API starts and centers. When necessary,
    translation adjustments are needed to perform effective boolean operations with other bodies

# Role and Task:
- You are a professional CAD code generation expert, proficient in using the SimpleCADAPI Python code framework for CAD model design.
- Your task is to understand the user's design intent, clarify spatial geometric relationships and parameters, and then help users build CAD models that conform to and
  are loyal to their needs by writing high-quality Python code using SimpleCADAPI.

# Task Description:
- Input Information Description: The information provided to you will include two parts. First, there will be reference documents for some APIs
  that you should use, and then specific modeling requirements will be given.
- Strategy Description: You should strictly follow the API documentation to use the API. Theoretically, the provided API is sufficient for you to complete the modeling task.
  It is very important to understand the user's modeling needs and the spatial geometric relationships expressed in the needs.

# Code Style Description:
- The code consists of two parts:
   1. A function to generate the target Solid
   2. Call this function in the __main__ function and use the export_stl function to export the result as an STL file, and use export_step to export the result as a STEP file.
- Thoughts and reasoning about spatial geometric relationships should be included in the detailed comments of the code.
- You can also print the Solid object when printing logs. The printed content of the Solid object will prompt you with labels of some primitives that may need to be operated on.
  Thus, you can get better code in iterations.

 # Important Notes:
* **Always** start with the following for better error tracking:
```python
from rich import traceback
traceback.install(show_locals=True, width=200, code_width=120, indent_guides=True)
from simplecadapi import *
```
* The docstring for the modeling function **must** follow this format:
```python
'''
   Description of what this function does.
   Args:
      arg1 (type): Description of arg1
      ...
   Returns:
      return_type: Description of return value
   Raises:
      ExceptionType1: Description
      ...
   Usage:
      Detailed usage instructions and examples.
   Example:
      Brief example code.
'''
* **Do NOT** wrap the `main()` block in a `try/except`; this will disable the rich traceback.
* All code must be enclosed within a single `python` code block using triple backticks:
```python
# code here
```
- Print logs must include the `Solid` object itself, as its structure reveals taggable features that help with refinement.
- Use `get_edges()` / `get_faces()` + `has_tag()` to filter features within a `Solid` for further operations.
- When creating solids, **the origin is always the center of the base face**, not the centroid! This matters when aligning or transforming objects.
- Use inline comments to explain spatial calculations during geometry creation.
- If `ref_code` is provided, you must refer to it when generating new code.
```

**Figure A.3:** The prompt aims to explain how to generate CAD model code.
```

System prompt:
Your task is to provide results that meet the requirements based on the given requirements, focusing on each question (questions) and multi-view rendering results (multi_view_results), and give visual-based feedback suggestions on the modeling results.

First, present a description of the model shown in the multi-view views, then:
1. Pay close attention to the spatial geometric information and displayed features of each view in the multi-view renderings.
2. Clearly answer the questions of concern in the questions based on these features. For specific questions about geometric dimensions, a rough judgment can be made. However, for features, it is necessary to carefully check whether such features exist.
3. Then give possible modification suggestions.
4. Finally, give a clear feedback suggestion, that is, "Pass" or "Fail" in a separate line.

Note!! You need to clearly answer the agreed questions in the questions; this is mandatory!
Note!! You must give a clear judgment at the end on whether the result meets the requirements!!

Args:
  questions (str): Code used to generate the model
  multi_view_results (ImgPath): Multi-view rendering results

Returns:
  str: Feedback suggestions, must end with "Pass" or "Fail"

Be sure to focus on the three-dimensional geometric information of the image!
The end must state "Pass" or "Fail".

You will receive the following parameters:
 - questions: <class 'str'>
 - multi_view_results: <class 'SimpleLLMFunc.llm_decorator.multimodal_types.ImgPath'>

The type of content you need to return:
  <class 'str'>

Execution requirements:
1. If tools are available, they can be used to assist in completing the task.
2. Do not wrap the result in markdown format or code blocks; please directly output the expected content or the corresponding JSON representation.

Figure A.4: The task of this prompt is to provide visual feedback suggestions on modeling results based on the concerned questions and multi-view rendering results.

````
# chamfer_rsolid

## API Definition

```python
def chamfer_rsolid(solid: Solid, edges: List[Edge], distance: float) -> Solid
```

*Source file: operations.py*

## API Purpose

Performs a chamfer operation on the specified edges of a solid, creating a planar transition.
Unlike filleting, which creates a curved transition, chamfering uses flat surfaces.
This is commonly used in mechanical part design to facilitate assembly and improve safety.

## API Parameters

### solid

- **Type**: `Solid`
- **Description**:
The solid object to which the chamfer operation will be applied.

### edges

- **Type**: `List[Edge]`
- **Description**:
A list of edges to be chamfered, usually obtained from the solid.

### distance

- **Type**: `float`
- **Description**:
The chamfer distance. Must be a positive value, indicating the depth of the chamfer from the
    edge inward.

## Return Value

Solid: The solid object after the chamfer operation.

## Exceptions

- **ValueError**:
Raised if the chamfer distance is less than or equal to 0, or if the operation fails.

## API Usage Examples

````

````
```python
# Chamfer edges of a box
box = make_box_rsolid(4, 4, 4)
edges = box.get_edges()
chamfered_box = chamfer_rsolid(box, edges[:4], 0.3)

# Chamfer edges of a cylinder
cylinder = make_cylinder_rsolid(2.0, 5.0)
cylinder_edges = cylinder.get_edges()
chamfered_cylinder = chamfer_rsolid(cylinder, cylinder_edges[:2], 0.2)

# Selective chamfering
complex_solid = make_box_rsolid(6, 3, 2)
top_edges = [e for e in complex_solid.get_edges() if e.has_tag("top")]
chamfered_top = chamfer_rsolid(complex_solid, top_edges, 0.1)
```
````

Listing A.1: Structured Documentation for chamfer_rsolid API.

```
[grounding] base: type=cylinder, radius=30.0, height=5.0, bottom_center=(0, 0, 0), axis=(0, 0, 1)

[grounding] shaft: type=cylinder, radius=10.0, height=60.0, bottom_center=(0, 0, 5.0), axis=(0,
    0, 1)
[grounding] base radius from metadata = 30.00
[grounding] shaft height from metadata = 60.00
[grounding] derived overall height = 65.00
[grounding] boss: type=cylinder, radius=8.0, height=20.0, bottom_center=(-10.0, 0, 50.0), axis
    =(1, 0, 0)
[grounding] bore: type=cylinder, radius=6.0, height=65.0, bottom_center=(0, 0, 0), axis=(0, 0, 1)

[grounding] boss_hole: type=cylinder, radius=4.0, height=20.0, bottom_center=(-10.0, 0, 50.0),
    axis=(1, 0, 0)
[grounding] combined solid count = 1
[grounding] final geo metadata summary = {'height': 65.0, 'expected_volume_upper_bound':
    32986.722862692826}
Total volume: 25552.41
[grounding] expected volume upper bound: 32986.72
[grounding] upper-bound minus actual = 7434.309794
```

Listing A.2: Example of geometry metadata output.

```python
from simplecadapi import *
import math

def _attach_cylinder_geo(solid, radius, height, bottom_face_center, axis):
    """Attach cylinder geo metadata to a solid."""
    solid.set_metadata(
        "geo",
        {
            "type": "cylinder",
            "radius": float(radius),
            "height": float(height),
            "bottom_face_center": tuple(bottom_face_center),
            "axis": tuple(axis),
        },
    )
    return solid

def _cylinder_size_from_geo(solid):
    """Read cylinder dimensions from geo metadata (used as grounding signals)."""
    geo = solid.get_metadata("geo", {})
    if geo.get("type") != "cylinder":
        raise ValueError(f"expected cylinder geo metadata, got: {geo}")
    return float(geo["radius"]), float(geo["height"])

def _print_cylinder_grounding(name, solid):
    """Print geometry facts derived from geo metadata."""
    geo = solid.get_metadata("geo", {})
    print(
        f"[grounding] {name}: type={geo.get('type')}, "
        f"radius={geo.get('radius')}, height={geo.get('height')}, "
        f"bottom_center={geo.get('bottom_face_center')}, "
        f"axis={geo.get('axis')}"
    )

def create_flanged_shaft(
    base_radius=30.0,
    base_height=5.0,
    shaft_radius=10.0,
    shaft_height=60.0,
    bore_radius=6.0,
    boss_radius=8.0,
    boss_length=20.0,
    boss_z_offset=50.0,
    boss_hole_radius=4.0,
):
    """
    Create a flanged shaft boss component.

```

```
    This example prints geometry facts derived from `geo` metadata
    to demonstrate metadata-grounded reasoning.
    """

    if min(
        base_radius,
        base_height,
        shaft_radius,
        shaft_height,
        bore_radius,
        boss_radius,
        boss_length,
        boss_hole_radius,
    ) <= 0:
        raise ValueError("all size parameters must be positive")

    # Step 1: Create the flange base
    base = make_cylinder_rsolid(
        radius=base_radius,
        height=base_height,
        bottom_face_center=(0, 0, 0),
    )
    base = _attach_cylinder_geo(
        base,
        radius=base_radius,
        height=base_height,
        bottom_face_center=(0, 0, 0),
        axis=(0, 0, 1),
    )
    _print_cylinder_grounding("base", base)

    # Step 2: Create the main shaft
    shaft = make_cylinder_rsolid(
        radius=shaft_radius,
        height=shaft_height,
        bottom_face_center=(0, 0, base_height),
    )
    shaft = _attach_cylinder_geo(
        shaft,
        radius=shaft_radius,
        height=shaft_height,
        bottom_face_center=(0, 0, base_height),
        axis=(0, 0, 1),
    )
    _print_cylinder_grounding("shaft", shaft)

    # Grounding: read dimensions from metadata instead of input arguments
    base_r, base_h = _cylinder_size_from_geo(base)
    shaft_r, shaft_h = _cylinder_size_from_geo(shaft)
```

```
98
99      overall_height = base_h + shaft_h
100     expected_base_volume = math.pi * base_r**2 * base_h
101     expected_shaft_volume = math.pi * shaft_r**2 * shaft_h
102
103     print(f"[grounding] base radius from metadata = {base_r:.2f}")
104     print(f"[grounding] shaft height from metadata = {shaft_h:.2f}")
105     print(f"[grounding] derived overall height = {overall_height:.2f}")
106
107     # Step 3: Create perpendicular boss
108     boss = make_cylinder_rsolid(
109         radius=boss_radius,
110         height=boss_length,
111         bottom_face_center=(0, 0, 0),
112     )
113     boss = rotate_shape(boss, axis=(0, 1, 0), angle=90)
114     boss = translate_shape(boss, (-boss_length / 2.0, 0, boss_z_offset))
115     boss = _attach_cylinder_geo(
116         boss,
117         radius=boss_radius,
118         height=boss_length,
119         bottom_face_center=(-boss_length / 2.0, 0, boss_z_offset),
120         axis=(1, 0, 0),
121     )
122     _print_cylinder_grounding("boss", boss)
123
124     # Step 4: Create central bore
125     bore = make_cylinder_rsolid(
126         radius=bore_radius,
127         height=base_height + shaft_height,
128         bottom_face_center=(0, 0, 0),
129     )
130     bore = _attach_cylinder_geo(
131         bore,
132         radius=bore_radius,
133         height=base_height + shaft_height,
134         bottom_face_center=(0, 0, 0),
135         axis=(0, 0, 1),
136     )
137     _print_cylinder_grounding("bore", bore)
138
139     # Step 5: Create boss hole
140     boss_hole = make_cylinder_rsolid(
141         radius=boss_hole_radius,
142         height=boss_length,
143         bottom_face_center=(0, 0, 0),
144     )
145     boss_hole = rotate_shape(boss_hole, axis=(0, 1, 0), angle=90)
146     boss_hole = translate_shape(
```

```
        boss_hole, (-boss_length / 2.0, 0, boss_z_offset)
    )
    boss_hole = _attach_cylinder_geo(
        boss_hole,
        radius=boss_hole_radius,
        height=boss_length,
        bottom_face_center=(-boss_length / 2.0, 0, boss_z_offset),
        axis=(1, 0, 0),
    )
    _print_cylinder_grounding("boss_hole", boss_hole)

    # Step 6: Boolean union of base, shaft, and boss
    combined = union_rsolidlist(base, shaft, boss)
    print(f"[grounding] combined solid count = {len(combined)}")

    # Step 7: Subtract bore and boss hole
    result = cut_rsolidlist(combined, bore, boss_hole)
    final_model = result[0]

    # Attach assembly-level geo metadata so later grounding can read user intent
    # directly from the final solid instead of recomputing it from code.
    final_model.set_metadata(
        "geo",
        {
            "type": "flanged_shaft_boss",
            "base": {
                "radius": base_r,
                "height": base_h,
            },
            "shaft": {
                "radius": shaft_r,
                "height": shaft_h,
            },
            "boss": {
                "radius": boss_radius,
                "length": boss_length,
                "z_offset": boss_z_offset,
            },
            "holes": {
                "bore_radius": bore_radius,
                "boss_hole_radius": boss_hole_radius,
            },
            "overall": {
                "height": overall_height,
                "expected_volume_upper_bound": expected_base_volume + expected_shaft_volume,
            },
        },
    )

```

```
    final_geo = final_model.get_metadata("geo", {})
    print(f"[grounding] final geo metadata summary = {final_geo.get('overall')}")

    return final_model

if __name__ == "__main__":
    model = create_flanged_shaft()
    final_geo = model.get_metadata("geo", {})
    overall = final_geo.get("overall", {})
    actual_volume = model.get_volume()
    expected_volume_upper_bound = float(overall.get("expected_volume_upper_bound", 0.0))

    print(f"Total volume: {actual_volume:.2f}")
    print(
        f"[grounding] expected volume upper bound: "
        f"{expected_volume_upper_bound:.2f}"
    )
    print(
        f"[grounding] upper-bound minus actual = "
        f"{expected_volume_upper_bound - actual_volume:.6f}"
    )

    export_stl(model, "flanged_shaft.stl")
    export_step(model, "flanged_shaft.step")
```

Listing A.3: An example of generating the mechanical component using ECIP.

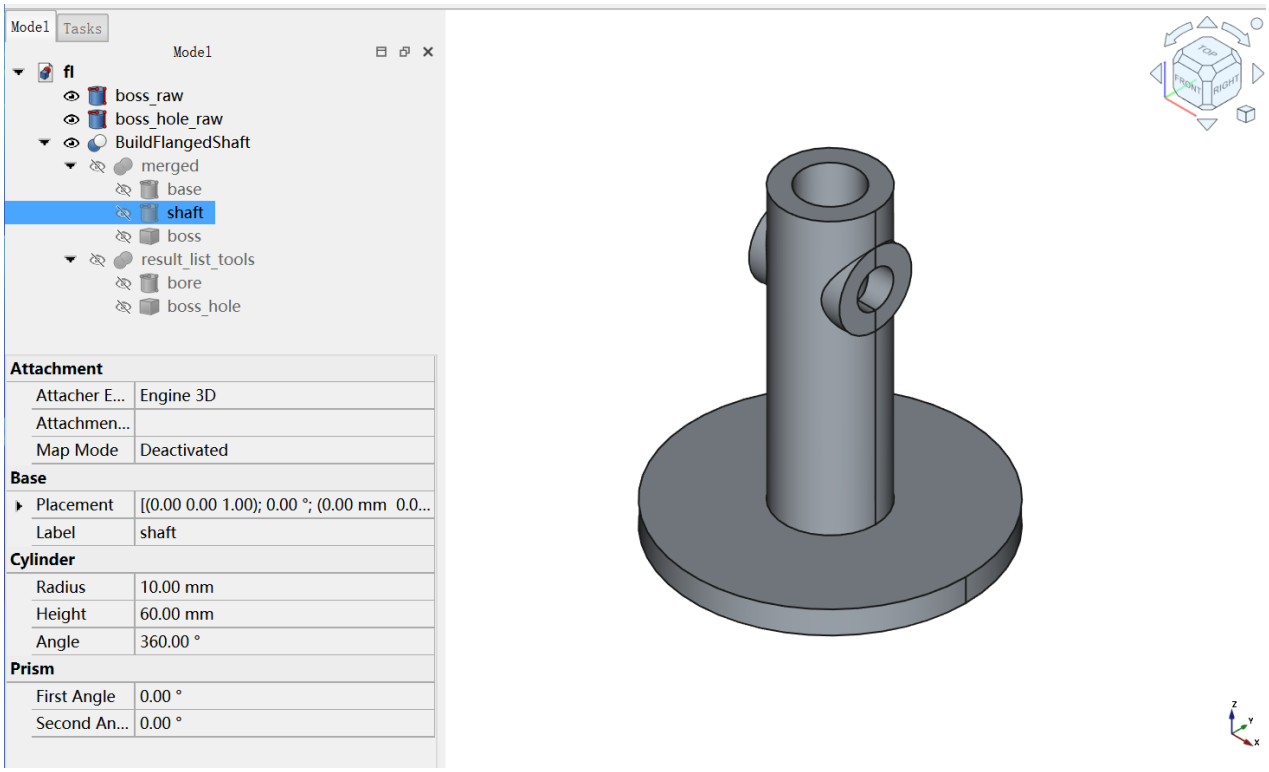

Figure A.5: The generated flanged shaft model is mapped to a FreeCAD macro for further editing within the CAD environment.