SKETCH–SOLVER: DECOUPLING CONSTRUCTION AND PARAMETERIZATION FOR NATURAL-LANGUAGE CAD PROGRAM SYNTHESIS

Anonymous authors
Paper under double-blind review

ABSTRACT

Natural-language CAD generation is commonly formulated as end-to-end translation, where a single decoder interleaves construction logic and numerical values within the same program sequence. This formulation imposes two distinct burdens: parameters must be chosen before the construction is complete, while repeated and related numerical occurrences are treated as independent predictions even when they correspond to fewer underlying degrees of freedom. Consequently, generation is vulnerable to structural errors and must recover parametric consistency through repeated autoregressive agreement. We propose **Sketch–Solver**, a framework that reorganizes both the generation process and the numerical representation. Its *construction–parameter factorization* separates construction from parameterization: *Sketch* generates a symbolic construction skeleton, and *Solver* subsequently instantiates the completed structure conditioned on both the instruction and the skeleton. Within this factorization interface, a *constraint-aware reparameterization* collapses repeated quantities into shared parameters and expresses selected proportional relations as executable dependencies, reducing unnecessary numerical decisions and yielding a canonical parameterization. Deterministic composition then produces the final executable CAD program. Experiments across multiple backbones and corpus-quality regimes show that Sketch–Solver improves synthesis robustness; controlled ablations separately identify the benefit of completed-construction conditioning and show that shared parameter names sharply reduce invalid and severe failures. The same framework also enables more reliable parameter-only editing through skeleton reuse. Together, these results show that decoupling construction from parameterization offers a new representation for CAD generation and an extensible interface for future methods.

1 INTRODUCTION

Computer-Aided Design (CAD) plays a central role in engineering and manufacturing (Wu et al., 2021). Unlike a mesh, a CAD model is an executable and editable *program* whose construction history and dimensions remain available for later modification. Natural-language CAD authoring must therefore recover not only plausible geometry, but also a valid construction, its numerical instantiation, and the parameter relationships required for controlled modification.

Recent advances in large language models (LLMs) have enabled natural-language CAD generation as either structured construction sequences (Wu et al., 2021; Khan et al., 2024; Govindarajan et al., 2026) or executable code (Xie & Ju, 2025; Li et al., 2025a; Guan et al., 2025). Despite these representational differences, most approaches predict the complete CAD program as a single autoregressive target, interleaving construction statements with numerical values and reproducing repeated dimensions at each use. Existing work improves this process through larger corpora, stronger decoders, or iterative feedback and repair (Li et al., 2026; Alrashedy et al., 2025; Wang et al., 2025; An et al., 2026). We instead revisit the generation interface and numerical representation themselves.

A CAD program contains two qualitatively different kinds of decisions: its *construction* determines which features and operations create the part, whereas its *parameterization* assigns the values that

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

instantiate them. The difficulty is not merely that these decisions differ in type. In a single autoregressive stream, numerical values must often be committed before the construction is complete, even though later operations may determine how those values are used or related. Parameter prediction is therefore made under incomplete structural context, while errors in construction and numerical instantiation can compound along the same decoding trajectory.

We propose **Sketch–Solver**, a framework that reorganizes both the generation process and the numerical representation. Its *construction–parameter factorization* separates the two prediction problems: *Sketch* generates a symbolic construction skeleton whose numerical slots reference named parameters, and *Solver* then instantiates this completed structure conditioned on both the instruction and the skeleton. Deterministic composition of the parameter block and skeleton recovers the final executable CAD program.

Factorization alone does not determine how the numerical interface should be represented. Assigning an independent variable to every numerical occurrence would merely relocate the redundancy of literal generation into Solver, because many occurrences correspond to fewer effective numerical degrees of freedom: values may be reused across sites or related by simple dependencies. We therefore introduce *constraint-aware reparameterization*, which collapses repeated, type-compatible occurrences into shared parameters and expresses selected proportional relations as executable dependencies. Factorization thus changes *when and under what context* parameters are generated, whereas reparameterization changes *which numerical quantities* must be predicted independently, allowing parametric consistency to be encoded in the program rather than recovered through repeated agreement between distant tokens.

The same interface also supports parameter-only editing by reusing the predicted skeleton when the construction is intended to remain fixed. To ensure reliable supervision and evaluation, we audit the inherited CadQuery corpus against independently reconstructed reference geometry, construct fidelity-controlled data tiers, and verify each transformed target before training. Controlled experiments evaluate the framework across model families and corpus-quality regimes, isolate its two mechanisms through ablations, analyze long-span value preservation, and test skeleton reuse under numerical edits.

Our contributions are:

- We introduce **Sketch–Solver**, whose *construction–parameter factorization* separates symbolic construction from numerical instantiation, allowing Solver to condition on the completed skeleton.
- We introduce **constraint-aware reparameterization**, which reduces redundant numerical decisions through shared parameters and explicit proportional dependencies.
- We audit the inherited CadQuery corpus and construct fidelity-controlled data tiers with verified canonical targets for reliable training and evaluation.
- We provide controlled evaluations across model families and corpus-quality regimes, mechanism-level ablations and long-span analysis, and a parameter-only editing benchmark for skeleton reuse.

2 Related Work

CAD representations and language-conditioned program generation. Learning-based CAD generation has used CSG, B-rep, and ordered construction-history representations (Du et al., 2018; Yu et al., 2022; Jayaraman et al., 2022; Xu et al., 2024; Wu et al., 2021). Sequence-based methods such as DeepCAD and Text2CAD model sketch-and-extrude histories (Wu et al., 2021; Khan et al., 2024), while SkexGen factorizes topology, geometry, and extrusion variation into separate latent codebooks (Xu et al., 2022). More recent language-conditioned systems generate structured command sequences, parametric code, or executable CAD programs (Li et al., 2025a; Govindarajan et al., 2026; Xie & Ju, 2025; Guan et al., 2025). These representations factorize geometric or construction structure in different ways, but executable-code generation still typically interleaves construction operations with their numerical arguments. Sketch–Solver instead generates construction and numerical instantiation as separate code fragments that compose deterministically into an executable program. Additional CAD representations and generation systems are reviewed in Appendix A.

Structure-first synthesis and parametric consistency. Resolving structure before low-level details has precedents in program synthesis by sketching (Solar-Lezama, 2008) and coarse-to-fine semantic parsing (Dong & Lapata, 2018). Classical program sketching completes holes in a programmer-provided partial program, while coarse-to-fine decoding predicts an abstract structure before filling omitted arguments. Sketch–Solver adapts this principle to language-conditioned CAD: its intermediate skeleton is predicted from the instruction, provides the symbolic construction interface for subsequent numerical instantiation, and can share named parameters across distant uses. Related work on parametric CAD uses constraint-solver feedback to generate relations that better preserve design intent under modification (Casey et al., 2025). Our representation conservatively encodes repeated quantities and selected dependencies supported by the observed CadQuery program, rather than a complete latent geometric constraint system.

Feedback, editing, and evaluation. Recent CAD systems improve initial predictions through visual feedback, verification, self-refinement, or interactive tool use (Wang et al., 2025; Alrashedy et al., 2025; Li et al., 2026; Gong et al., 2026; Fan et al., 2026; An et al., 2026). CAD-Editor instead modifies localized regions of an existing CAD sequence (Yuan et al., 2025). Our parameter-only editing mode is complementary: when construction is intended to remain fixed, it reuses the predicted skeleton and regenerates only its parameterization. Recent benchmarks also broaden text-to-CAD evaluation toward more complex or functionally tested designs (Wang et al., 2026; Mallis et al., 2026); our experiments instead isolate the proposed formulation under matched data, backbone, and decoding, together with controlled construction-preserving numerical edits.

3 METHOD

Sketch–Solver rests on two complementary design principles. *Construction–parameter factorization* changes when and under what context numerical decisions are made, while *constraint-aware reparameterization* changes which quantities must be predicted independently. We describe each before corpus construction, execution, and editing.

3.1 CONSTRUCTION–PARAMETER FACTORIZATION

Given an instruction T, our goal is an executable CadQuery program P (CadQuery Developers, 2026). We represent it as (S, V): a *skeleton* S specifies the construction and references named numerical parameters, while a *parameter block* V defines those parameters through literals or arithmetic expressions. Here *Sketch* denotes the stage that produces this symbolic program skeleton, rather than a conventional two-dimensional CAD sketch.

We factorize generation into two conditional models,

$$p_{\theta,\phi}(S, V \mid T) = p_\theta(S \mid T)\, p_\phi(V \mid u(T, S)), \qquad P = M(S, V), \tag{1}$$

where u is a deterministic interface formatter and M composes the two code fragments. Figure 1(a) shows this factorization. Sketch first determines the complete construction; Solver then generates its parameterization conditioned on both T and S. We instantiate the two stages with separately trained language models. Thus, every Solver decision is made with access to the completed skeleton rather than only an earlier program prefix. This factorization specifies the prediction order but not which quantities should be independent; the next design determines that representation.

3.2 CONSTRAINT-AWARE REPARAMETERIZATION

Giving every numerical occurrence an independent name would merely move literal redundancy into V. We instead collapse repeated occurrences into shared parameters, then organize them into independent roots and dependent quantities. Figure 1(b) summarizes these two operations. The transformation is induced deterministically from source code without an LLM annotator.

Let $\mathcal{I}$ denote the selected numerical occurrences and $\eta : \mathcal{I} \to \Theta$ map each occurrence to a parameter name. Replacing them yields $S = E(P, \eta)$. If $\eta(i) = \eta(j)$, both sites read the same binding. Equality is therefore encoded in the skeleton once Sketch emits the shared references, rather than recovered through independent literals (Appendix C.5).

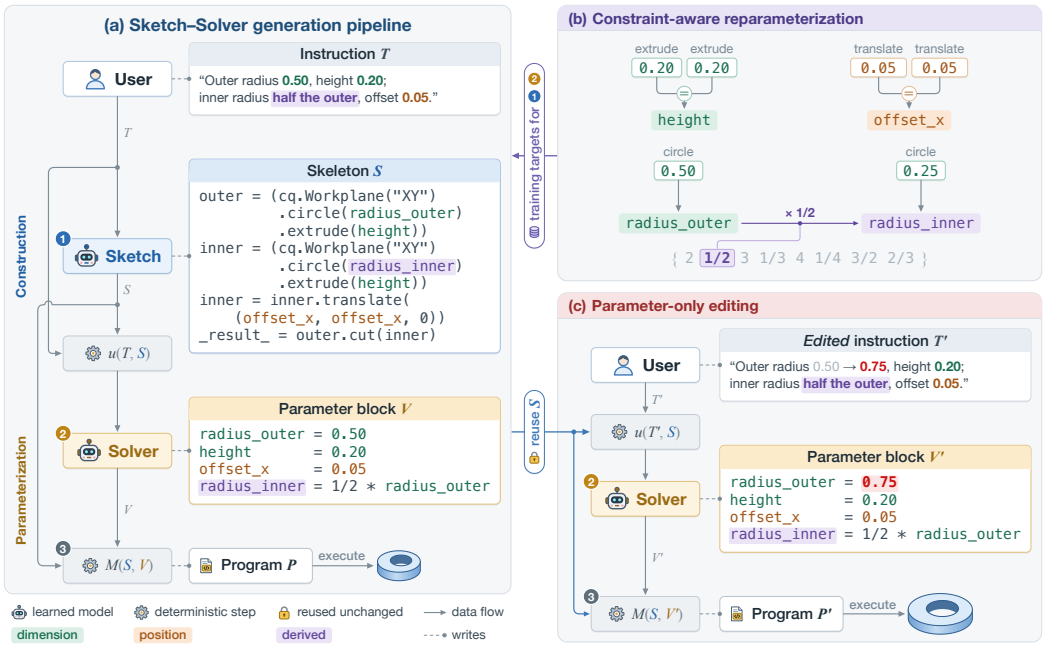


Figure 1: **Sketch–Solver overview. (a)** Sketch maps instruction T to construction skeleton S; Solver predicts parameter block V from $u(T, S)$; and deterministic composition produces executable program $P = M(S, V)$. **(b)** Constraint-aware reparameterization shares type-compatible equal values and writes selected ratios as executable dependencies, reducing independent numerical decisions. **(c)** Parameter-only editing reuses S, skips Sketch, and predicts V' from the revised instruction T' before recomposition into P'.

Typed equality sharing. Equality sharing is restricted to type-compatible occurrences to avoid coupling numerically equal but functionally unrelated quantities. Static analysis identifies parameterizable numerical values and assigns coarse types such as dimension, position, angle, and scale (Table 10 in Appendix C.2). The implementation uses identifier semantics first and CAD operation/argument context as a fallback. Within a type, occurrences with values differing by less than 10^{-6} share a name, yielding a conservative parameterization consistent with the observed program. A shared extrusion height, for example, makes the outer body and its cutting tool consume one assignment rather than separately generated height literals.

Sparse proportional dependencies. After equality sharing, we induce only a small family of common, low-complexity multiplicative relations. This finite vocabulary limits accidental symbolic fits in the quantized source data. For dimensional parameters whose observed values satisfy $a \simeq rb$, we consider

$$r \in \left\{2, \tfrac{1}{2}, 3, \tfrac{1}{3}, 4, \tfrac{1}{4}, \tfrac{3}{2}, \tfrac{2}{3}\right\}, \qquad |a - rb| \le 10^{-9} \max(|a|, |b|, 10^{-12}). \tag{2}$$

These relations form a sparse candidate graph, which we canonicalize into an executable forest. Parameters numerically stated in the instruction are preferred as roots; the remaining components follow a fixed extraction and traversal order. A separate rule may attach an independent position to half of a dimension. This deterministic construction resolves ambiguous edges, prevents cycles, and yields a topological assignment order; Appendix C.3 gives the complete procedure.

The parameter block contains literal assignments for independent roots and expressions for dependent parameters, such as `inner_radius = 1/2 * outer_radius`. Equality is carried by repeated names in S, whereas proportional dependence is carried by expressions in V. Together they reduce the numerical interface from independent occurrences to roots and explicit dependencies. The channels remain operationally distinct: removing ratio expressions leaves skeleton-level sharing intact, and an arithmetic relation remains a Solver prediction unless the corresponding expression is generated.

3.3 VERIFIED CORPUS, SUPERVISION, AND EXECUTION

Source-corpus audit. The inherited Text-to-CadQuery corpus (Xie & Ju, 2025) pairs translated CadQuery programs with source constructions, but buildability does not ensure geometric fidelity. We independently replay the source construction, execute the translation, and compare their normalized meshes. Progressively relaxed distance thresholds define four nested tiers—Verified-CQ@CD5, Verified-CQ@CD10, Verified-CQ@CD15, and Verified-CQ@CD20—where Verified-CQ denotes our independently audited Text-to-CadQuery corpus; one persisted verdict per record makes membership reproducible. We further retain a reparameterized target only if exact-arithmetic reconstruction recovers its source values, the composed program executes with every name resolved and one final-result binding, and its mesh agrees with the source in volume and surface area. Statistics and discrepancy analysis appear in Appendix B.

Training. Each accepted (T, P) produces aligned targets (T, S) for Sketch and $(u(T, S), V)$ for Solver. Both models are initialized from the same backbone and fine-tuned separately with assistant-token cross-entropy:

$$\mathcal{L}(\theta, \phi) = -\sum_{(T,S,V)} \left[\log p_\theta(S \mid T) + \log p_\phi(V \mid u(T, S))\right]. \tag{3}$$

The formatter u supplies the instruction, complete skeleton, and unresolved names in first-occurrence order. Names are recovered from S, so Sketch does not generate a separate parameter inventory, and the same formatter defines the hand-off during training and inference. Appendix H compares this AST-mediated interface with explicit inventory generation.

Inference. At inference, Sketch emits the construction skeleton and Solver generates the corresponding assignment block. After deterministic text cleanup, $M(S, V)$ places the import preamble and V before S, and the resulting assignments are evaluated through ordinary Python execution. Composition is deterministic; the audit and verification procedures above construct the corpus and supervision but are absent from the inference path. Appendix D documents the implemented recovery rules and execution budgets.

3.4 PARAMETER-ONLY EDITING THROUGH SKELETON REUSE

Construction–parameter factorization also exposes a reusable editing interface. After an initial generation, retain its predicted skeleton $\hat{S}_0$. For a revised instruction T' that changes numerical specifications while retaining the construction, generate

$$\hat{V}' \sim p_\phi(\cdot \mid u(T', \hat{S}_0)), \qquad \hat{P}' = M(\hat{S}_0, \hat{V}'). \tag{4}$$

The edit invokes only Solver and uses the model's own earlier skeleton. Figure 1(c) shows this path, which skips Sketch. For example, increasing an outer radius while retaining a hollow-cylinder construction requires new assignments, while the extrusion and subtraction statements remain fixed. Our editing study contrasts this reuse path with regenerating the complete program from the revised instruction.

Skeleton reuse preserves the construction code and all shared parameter references encoded in it, while Solver regenerates the complete assignment block. Shared names therefore continue to couple their uses without asking Sketch to reproduce those references again; arithmetic relations in V remain predictions rather than invariants. This mode applies to requests that preserve the construction. Structural revisions require rerunning Sketch.

4 EXPERIMENTS

We evaluate Sketch–Solver through cross-paper comparison, controlled synthesis across corpus tiers and model families, ablations of its two design principles, and parameter-only editing. The original Text-to-CadQuery split supports comparison with prior work, while a shared verified set supports the controlled studies.

4.1 EXPERIMENTAL SETUP

Benchmarks and splits. The main comparison uses the original Text-to-CadQuery split (Xie & Ju, 2025), containing 99,236/5,513/5,514 train/validation/test records. Our audit finds extensive geometric translation error in this corpus, which is not filtered by independent fidelity; we therefore use it for protocol continuity and reserve controlled comparisons for the verified corpus. Our controlled studies use Verified-CQ, the audited corpus introduced in Sec. 3: 36,642, 47,027, 54,232, and 58,813 records pass the independent-reference thresholds $\mathrm{CD} \times 10^3 < 5, 10, 15, 20$, defining the nested Verified-CQ@CD5, Verified-CQ@CD10, Verified-CQ@CD15, and Verified-CQ@CD20 tiers. They contain 33,001–53,035 training examples. Unless a table specifies otherwise, controlled runs are evaluated on the same 2,926 validation IDs from the widest verified tier. Direct and Sketch–Solver targets within every tier are aligned example by example.

Models and baselines. The primary backbone is Qwen3-4B-Instruct (Qwen Team, 2025). We compare direct CadQuery generation with Sketch–Solver under the same training examples and optimization recipe. Qwen3.5-2B (Qwen Team, 2026) and Ministral 3 3B (Mistral AI, 2025) test whether the behavior transfers across model families. We also evaluate DeepSeek-v4-Flash, DeepSeek-v4-Pro, GPT-5.6-Luna, and GPT-5.6-Terra through a uniform four-shot Direct prompt. These prompted models are external reference points; the matched Direct–Sketch–Solver pairs test the proposed formulation. Recent systems such as CAD-Coder, CADmium, CAD-Llama, and Seek-CAD (Guan et al., 2025; Govindarajan et al., 2026; Li et al., 2025a; 2026) use different data, representations, or feedback protocols and do not report results under the released Text-to-CadQuery test protocol, so they are discussed in Sec. 2 rather than mixed into Table 1.

Metrics. We report three metrics. Invalid rate (IR) is the percentage of programs that fail to produce a solid. Mean Chamfer Distance (CD) is the symmetric squared ℓ_2 distance between 10,000-point surface samples, multiplied by 10^3 and averaged over valid predictions (Fan et al., 2017; Xie & Ju, 2025). Following Text-to-CadQuery, each mesh is centered independently and scaled by its maximum bounding-box extent. Finally, FR_{100} is the percentage of all inputs that are invalid or have $\mathrm{CD} \geq 100$, exposing severe failures hidden by valid-only mean CD. Lower is better for all three metrics. Additional evaluator details appear in Appendix F.

Training and inference. All SFT rows use full-parameter bfloat16 supervised fine-tuning. The verified-corpus runs use two epochs, learning rate 10^{-5}, effective batch size 16, and assistant-token loss. Sketch and Solver are trained as separate stages using their aligned targets. Generation is greedy. Each composed program executes in a subprocess with a 10-second timeout and exports an STL for evaluation. Training and decoding configurations and exact prompts are provided in Appendices E and I.

4.2 MAIN SYNTHESIS RESULTS

Original Text-to-CadQuery split. Table 1 separates values reported by Text-to-CadQuery (Xie & Ju, 2025) on its released length-filtered test file from our evaluation of all 5,514 source records, including 5,472 with buildable references. The two blocks use different test populations and should not be read as a direct ranking. Our Direct and Sketch–Solver routes use aligned cleaned targets ending in `_result_` and the same execution and scoring harness. Following the published Qwen recipe, both are full-parameter fine-tuned for three epochs with learning rate 5×10^{-5} and effective batch size 12. Sketch–Solver lowers IR on both Qwen backbones, while the geometric metrics are tied or mixed. This muted difference is consistent with the tier study: gains are clearest under higher-fidelity supervision and narrow as less faithful annotations are admitted. Published baselines do not report FR_{100}.

Controlled comparison on the verified corpus. Table 2 compares prompted API models with matched SFT pairs on the same 2,926 inputs. The SFT models are trained on Verified-CQ@CD15, and each Direct–Sketch–Solver pair shares its data and optimization recipe. Sketch–Solver consistently improves IR and the severe-failure rate on both Qwen backbones, with smaller gains in valid-only mean CD; the two Ministral formulations are effectively tied. Appendix G.2 decomposes

Table 1: Results under the two original-split test protocols identified below.

Method	IR↓	Mean CD↓	FR_{100} ↓
Published: released length-filtered test file			
Text2CAD Transformer	3.50	26.42	–
Text-to-CadQuery, CodeGPT-small	9.40	13.52	–
Text-to-CadQuery, GPT-2 medium	15.60	12.57	–
Text-to-CadQuery, GPT-2 large	13.80	12.18	–
Text-to-CadQuery, Gemma3-1B	8.40	11.61	–
Text-to-CadQuery, Qwen2.5-3B	6.50	10.23	–
Text-to-CadQuery, Mistral-7B	1.32	11.84	–
Full source test under our shared harness			
Qwen3-4B Direct	0.93	13.92	5.03
Qwen3-4B Sketch–Solver (ours)	**0.68**	**13.91**	**4.59**
Qwen3.5-2B Direct	1.15	**13.42**	**4.84**
Qwen3.5-2B Sketch–Solver (ours)	**1.06**	14.26	5.26

invalid and severe failures across the two evaluation settings. The API rows provide external direct-generation reference points; evidence for the formulation comes from the controlled SFT pairs.

Table 2: Controlled comparison on 2,926 verified inputs; SFT models are trained on Verified-CQ@CD15.

Regime	Model	Formulation	IR↓	Mean CD↓	FR_{100} ↓
4-shot API	DeepSeek-v4-Flash	Direct	12.82	21.99	21.67
	DeepSeek-v4-Pro	Direct	4.78	**20.74**	13.81
	GPT-5.6-Luna	Direct	**2.73**	21.38	**12.54**
	GPT-5.6-Terra	Direct	5.33	21.37	14.29
SFT	Qwen3-4B	Direct	3.35	10.37	6.80
	Qwen3-4B	Sketch–Solver	**2.60**	**9.14**	**5.47**
SFT	Qwen3.5-2B	Direct	3.11	7.91	5.33
	Qwen3.5-2B	Sketch–Solver	**1.88**	**7.42**	**3.76**
SFT	Ministral 3 3B	Direct	1.20	**7.14**	**3.04**
	Ministral 3 3B	Sketch–Solver	**1.16**	7.26	3.11

4.3 Results across corpus-quality tiers

The verified tiers expose the same Direct–Sketch–Solver comparison under progressively broader corpus admission thresholds. Table 3 shows that Sketch–Solver lowers mean CD in all four tiers and reduces the full-set severe-failure rate in every tier. The reduction is largest in Verified-CQ@CD5 through Verified-CQ@CD15; at Verified-CQ@CD20, where the training corpus is largest but admits less faithful annotations, the two formulations become closer. The difficulty-stratified results in Appendix Table 12 locate the most consistent gains on medium-complexity parts, where all three metrics improve in every tier. The pattern indicates that the formulation benefits most from supervision that preserves the reference geometry. Because the tiers jointly change corpus fidelity and sample count, they characterize the corpus-quality frontier rather than isolate data efficiency.

Table 3: Robustness across nested corpus tiers with Qwen3-4B; S–S denotes Sketch–Solver.

		IR↓		Mean CD↓		FR_{100} ↓	
Tier	Train n	Direct	S–S	Direct	S–S	Direct	S–S
Verified-CQ@CD5	33,001	3.93	**3.04**	11.74	**10.79**	7.96	**6.02**
Verified-CQ@CD10	42,388	3.59	**2.56**	10.47	**9.69**	7.04	**5.33**
Verified-CQ@CD15	48,878	3.35	**2.60**	10.37	**9.14**	6.80	**5.47**
Verified-CQ@CD20	53,035	**3.21**	**3.21**	9.51	**9.14**	6.39	**5.98**

4.4 DISENTANGLING THE TWO DESIGN PRINCIPLES

The factorization study uses Verified-CQ@CD10, and the representation study uses Verified-CQ@CD5. Within each study, all variants use Qwen3-4B, the same training examples and optimization recipe, and the same 2,926 evaluation inputs.

Table 4: Construction–parameter factorization on Verified-CQ@CD10.

Variant	IR↓	Mean CD↓	FR_{100} ↓
Direct	3.59	10.47	7.04
Joint symbolic	3.76	12.13	7.72
Sketch–Solver	**2.56**	**9.69**	**5.33**

Table 5: Constraint-aware reparameterization on Verified-CQ@CD5.

Variant	IR↓	Mean CD↓	FR_{100} ↓
Sketch–Solver	**3.04**	**10.79**	**6.02**
No shared equality	8.44	10.92	11.45
No ratio expressions	3.18	11.01	6.46
Expanded relations	3.52	11.07	6.56

Completed-construction conditioning. Joint symbolic generation emits the same skeleton and assignment block as Sketch–Solver in one response, but underperforms Direct on all three metrics. Sketch–Solver predicts the same symbolic content across two stages and outperforms both alternatives. Its two outputs together are no shorter than Direct, locating the improvement in conditioning order rather than target length. Holding the target representation fixed, this gap isolates the benefit of completing construction before parameterization.

Shared-name equality. Removing shared-name equality replaces each repeated read with an independently named parameter while preserving the instantiated training values. The sharp increase in invalid and severe failures, despite little change in valid-only CD, identifies shared names as the main representation-level source of robustness.

Repeated-value preservation. At the framework level, we further measure whether a known repeated value is preserved as the distance between its uses increases. Sketch–Solver exceeds Direct by 5.5 percentage points below 60 characters and by 13.3 points at spans of 400 or more; its aggregate preservation rate is 75.0%, compared with 67.2% for Direct (Appendix G.3). This analysis measures factorization and representation together, while the controlled equality ablation above provides the mechanism-specific evidence for shared names.

Arithmetic dependencies. Removing the sparse proportional expressions causes a modest but consistent degradation. Expanding the relation vocabulary raises expression density from 3.1% to 20.8%, while diagnostic precision falls from 0.738 to 0.586 and synthesis does not improve (Appendices G.4 and G.5). Explicit dependencies are therefore most useful when induced conservatively.

4.5 PARAMETER-ONLY EDITING

Benchmark construction. Starting from the 2,926 verified validation parts, we edit $L \in \{1, 2, 3\}$ instruction-grounded values by a common factor $k \in \{0.5, 1.5, 2.0\}$. Selected parameters must participate in a shared read or dependency expression; simultaneous edits use disjoint dependency cones and distinct textual values. We rewrite the instruction and reference roots, reevaluate dependent assignments, and retain only cases that build, satisfy the requested scalar changes, preserve monitored values, and alter the geometry. Uniform scaling cases are excluded because normalized CD cannot distinguish them. Requiring all three factors at each level yields 3,366 tasks from 914 parts. Further construction details and certificates appear in Appendix G.6.

Table 6 reports IR and FR_{100} over all tasks; CD uses the common-valid intersection of the three routes within each backbone. Every reuse route consumes its model's own cached skeleton.

Sketch–Solver regeneration already outperforms Direct regeneration on both backbones, showing that the framework remains effective under modified specifications. Reusing the predicted skeleton improves further, most clearly on Qwen3.5-2B, because the predicted construction is retained while only its numerical instantiation is reconsidered. No route receives editing-specific training.

Table 6: Parameter-only editing on Verified-CQ@CD15 across 3,366 tasks.

Backbone	Route	CD↓	IR↓	FR_{100} ↓
Qwen3-4B	Direct regenerate	16.10	6.80	10.55
	Sketch–Solver regenerate	12.88	4.96	**7.13**
	Sketch–Solver reuse	**12.34**	**4.84**	**7.13**
Qwen3.5-2B	Direct regenerate	13.91	6.12	8.67
	Sketch–Solver regenerate	11.68	3.09	5.20
	Sketch–Solver reuse	**10.71**	**2.64**	**4.25**

At L1/L2/L3, common-valid CD (Direct regeneration / Sketch–Solver regeneration / reuse) is 13.85/11.73/11.26, 23.85/17.50/16.87, and 31.39/17.63/16.02 for Qwen3-4B; for Qwen3.5-2B, it is 12.02/10.10/9.35, 19.70/17.00/15.30, and 28.16/21.74/19.31. Thus, reuse retains its advantage as more values are changed on both backbones. Appendix Figure 2 visualizes these trends and reports the population at each level; Appendix G.7 provides qualitative examples.

5 Discussion and Limitations

Sketch–Solver gives the intermediate program two complementary roles. The joint symbolic ablation attributes the factorization gain to conditioning numerical decisions on a completed construction, while equality removal identifies shared names as the dominant dependency channel. The offline checker serves a separate purpose: it validates induced supervision and is absent from inference.

Skeleton reuse demonstrates the practical value of separating the two stages. For a parameter-only request, it retains the predicted construction while Solver regenerates its complete parameter block. This mode assumes that construction remains fixed: structural revisions require rerunning Sketch, and sufficiently large numerical changes can still cross geometric thresholds and alter solid topology.

Our evaluation has three main boundaries. Normalized CD is invariant to global translation and uniform scale, so the edit benchmark excludes global-similarity cases; its certificates cover constant-foldable scalars and bounding-box behavior rather than arbitrary dataflow. Controlled gains are strongest on Qwen3-4B and Qwen3.5-2B and nearly neutral on Ministral 3 3B. Corpus selection uses independent DeepCAD replay, whereas synthesis evaluation rebuilds the accepted CadQuery targets. Finally, the induced numerical relations are executable hypotheses rather than recovered ground-truth design intent; the expanded-relations study shows the value of conservative induction.

6 Conclusion

Sketch–Solver reformulates CAD program generation by explicitly separating construction from parameterization through an executable and reusable interface. Sketch first determines the symbolic construction, and Solver then parameterizes the completed structure. This ordering allows numerical generation to use the full construction, while shared names and executable dependencies represent repeated numerical relations directly rather than requiring their independent reproduction. Across the controlled verified-corpus studies, Sketch–Solver reduces severe synthesis failures on both Qwen backbones and remains nearly neutral on Ministral; on Qwen3-4B, it also preserves repeated values more reliably, especially across long spans. Once construction and parameterization are separated, the predicted construction can also be retained while only its parameterization is regenerated, enabling more reliable parameter-only editing without editing-specific training. Overall, these results support treating construction and parameterization as distinct, composable, and reusable components for robust and controllable CAD program synthesis.

AI use statement

In this work, we used generative AI tools to improve the clarity and readability of the manuscript and to create or edit software code. We did not use generative AI tools for any other research task requiring disclosure. The authors reviewed every AI-assisted manuscript edit to ensure that it

preserved the intended technical content, claims, and experimental results, and inspected and tested AI-assisted code for correctness. We take responsibility for the final content of this work.

Ethics statement

This work uses public CAD construction data and contains no human-subject or personally identifying information. Generated CAD programs can fail geometrically or structurally; the reported validation, failure metrics, and limitations should be considered before any manufacturing use.

Reproducibility statement

The main paper and appendices document the corpus audit and induction, verification gates, training and inference configurations, prompts, and evaluation protocol. We retain the processed corpus tiers, persisted verification decisions, model predictions, and analysis scripts used for the reported results. These artifacts are not publicly available at submission time and will be released upon publication.

References

Kamel Alrashedy, Pradyumna Tambwekar, Zulfiqar Zaidi, Megan Langwasser, Wei Xu, and Matthew Gombolay. Generating CAD code with vision-language models for 3D designs. In *International Conference on Learning Representations (ICLR)*, 2025.

Jiyuan An, Jiachen Zhao, Fan Chen, Liner Yang, Zhenghao Liu, Hongyan Wang, Weihua An, Meishan Zhang, and Erhong Yang. PR-CAD: Progressive refinement for unified controllable and faithful text-to-CAD generation with large language models. *arXiv preprint arXiv:2604.19773*, 2026.

Akshay Badagabettu, Sai Sravan Yarlagadda, and Amir Barati Farimani. Query2CAD: Generating CAD models using natural language queries. *arXiv preprint arXiv:2406.00144*, 2024.

CadQuery Developers. CadQuery: A Python parametric CAD scripting framework. Software, version 2.8.0, https://github.com/CadQuery/cadquery, 2026.

Evan Casey, Tianyu Zhang, Shu Ishida, John Roger Thompson, Amir Khasahmadi, Joseph George Lambourne, Pradeep Kumar Jayaraman, and Karl D. D. Willis. Aligning constraint generation with design intent in parametric CAD. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 8613–8622, 2025.

Li Dong and Mirella Lapata. Coarse-to-fine decoding for neural semantic parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 731–742, 2018.

Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. InverseCSG: Automatic conversion of 3D models to CSG trees. *ACM Transactions on Graphics*, 37(6):213:1–213:16, 2018. doi: 10.1145/3272127.3275006.

Fengxiao Fan, Jingzhe Ni, Xiaolong Yin, Sirui Wang, Xingyu Lu, Qiang Zou, Ruofeng Tong, Min Tang, and Peng Du. CADDesigner: Conceptual CAD model generation with a general-purpose agent. *Computer-Aided Design*, 198:104087, 2026. doi: 10.1016/j.cad.2026.104087.

Haoqiang Fan, Hao Su, and Leonidas J. Guibas. A point set generation network for 3D object reconstruction from a single image. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 605–613, 2017.

Yifei Gong, Xing Wu, Wenda Liu, and Kang Tu. TOOLCAD: Exploring tool-using large language models in text-to-CAD generation with reinforcement learning. *arXiv preprint arXiv:2604.07960*, 2026. arXiv:2604.07960.

Prashant Govindarajan, Davide Baldelli, Jay Pathak, Quentin Fournier, and Sarath Chandar. CADmium: Fine-tuning code language models for text-driven sequential CAD design. *Transactions on Machine Learning Research*, 2026. URL https://openreview.net/forum?id=lExqWvQht8. arXiv:2507.09792.

Yandong Guan, Xilin Wang, XiMing Xing, Jing Zhang, Dong Xu, and Qian Yu. CAD-Coder: Text-to-CAD generation with chain-of-thought and geometric reward. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. arXiv:2505.19713.

Pradeep Kumar Jayaraman, Joseph G. Lambourne, Nishkrit Desai, Karl D. D. Willis, Aditya Sanghi, and Nigel J. W. Morris. SolidGen: An autoregressive model for direct B-rep synthesis. *arXiv preprint arXiv:2203.13944*, 2022.

Soslan Kabisov, Vsevolod Kirichuk, Andrey Volkov, Gennadii Savrasov, Marina Barannikov, Anton Konushin, Andrey Kuznetsov, and Dmitrii Zhemchuzhnikov. CADReasoner: Iterative program editing for CAD reverse engineering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2026. Findings Track; arXiv:2603.29847.

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 (NeurIPS)*, volume 37, pp. 7552–7579, 2024.

Maksim Kolodiazhnyi, Denis Tarasov, Dmitrii Zhemchuzhnikov, Alexander Nikulin, Ilya Zisman, Anna Vorontsova, Anton Konushin, Vladislav Kurenkov, and Danila Rukhovich. Cadrille: Multimodal CAD reconstruction with online reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://openreview.net/forum?id=w2tnhhMbXv.

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 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 18563–18573, 2025a. arXiv:2505.04481.

Jing Li, Yihang Fu, and Falai Chen. DTGBrepGen: A novel B-rep generative model through decoupling topology and geometry. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 21438–21447, 2025b. arXiv:2503.13110.

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. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://openreview.net/forum?id=PzIc2TxhwN.

Dimitrios Mallis, Marco Wang, Ahmet Serdar Karadeniz, Elisa Ricci, Anis Kacem, and Djamila Aouada. Text-to-CAD evaluation with CADTests. *arXiv preprint arXiv:2605.07807*, 2026.

Mistral AI. Introducing Mistral 3. Mistral AI, https://mistral.ai/news/mistral-3/, 2025.

Ke Niu, Haiyang Yu, Zhuofan Chen, Mengyang Zhao, Teng Fu, Bin Li, and Xiangyang Xue. From intent to execution: Multimodal chain-of-thought reinforcement learning for precise CAD code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 8160–8167, 2026. doi: 10.1609/aaai.v40i10.37763.

Qwen Team. Qwen3 technical report, 2025.

Qwen Team. Qwen3.5: Towards native multimodal agents. Qwen Blog, https://qwen.ai/blog?id=qwen3.5, 2026.

Armando Solar-Lezama. Program synthesis by sketching. Technical Report UCB/EECS-2008-176, University of California, Berkeley, 2008.

Liang Wang, Heng Meng, Zekai Xiang, Jin Liu, Pingyi Zhou, Litao Chen, and Yongqiang Tang. Text2CAD-Bench: A benchmark for LLM-based text-to-parametric CAD generation. *arXiv preprint arXiv:2605.18430*, 2026.

Ruiyu Wang, Yu Yuan, Shizhao Sun, and Jiang Bian. Text-to-CAD generation through infusing visual feedback in large language models. In *International Conference on Machine Learning (ICML)*, 2025. arXiv:2501.19054.

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*, 40 (4):54:1–54:24, 2021. doi: 10.1145/3450626.3459818.

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 (ICCV)*, pp. 6772–6782, 2021.

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. Project page: https://github.com/Text-to-CadQuery/Text-to-CadQuery.

Xiang Xu, Karl D. D. 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 (ICML)*, 2022. arXiv:2207.04632.

Xiang Xu, Joseph Lambourne, Pradeep Jayaraman, Zhengqing Wang, Karl Willis, and Yasutaka Furukawa. BrepGen: A B-rep generative diffusion model with structured latent geometry. *ACM Transactions on Graphics*, 43(4):1–14, 2024. doi: 10.1145/3658129.

Fenggen Yu, Zhiqin Chen, Manyi Li, Aditya Sanghi, Hooman Shayani, Ali Mahdavi-Amiri, and Hao Zhang. CAPRI-Net: Learning compact CAD shapes with adaptive primitive assembly. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 11768–11778, 2022.

Bo Yuan, Zelin Zhao, Petr Molodyk, Bin Hu, and Yongxin Chen. Clarify before you draw: Proactive agents for robust text-to-CAD generation. *arXiv preprint arXiv:2602.03045*, 2026.

Yu Yuan, Shizhao Sun, Qi Liu, and Jiang Bian. CAD-Editor: A locate-then-infill framework with automated training data synthesis for text-based CAD editing. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, volume 267 of *Proceedings of Machine Learning Research*, pp. 73588–73603, 2025. URL https://proceedings.mlr.press/v267/yuan25g.html.

Zheyuan Zhou, Jiayi Han, Liang Du, Naiyu Fang, Lemiao Qiu, and Shuyou Zhang. CAD-Judge: Toward efficient morphological grading and verification for text-to-CAD generation. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1021–1025, 2026. doi: 10.1109/ICASSP55912.2026.11461978.

CONTENTS

A Additional Related Work **14**

B Corpus audit and tier selection **14**

C Corpus construction **15**

C.1 Source cleaning 15

C.2 Parameter induction 16

C.3 Ratio forest and parameter block 16

C.4 Verification gates 18

C.5 Properties of the induced targets 18

D Two-stage harness and post-processing **19**

D.1 Hand-off 19

D.2 Inference and composition 19

D.3 Post-processing for Direct generation 19

D.4 Train–inference shift 19

E Training and inference configurations **20**

E.1 Training 20

E.2 Decoding and sequence lengths 20

E.3 API evaluation setup 20

F Evaluation metrics **20**

G Additional results and diagnostics **21**

G.1 Synthesis by part difficulty 21

G.2 Failure decomposition 21

G.3 Repeated-value preservation 22

G.4 Representation-ablation diagnostics 23

G.5 Expanded-relations intervention 23

G.6 Editing by number and magnitude of changes 24

G.7 Qualitative parameter-only edits 25

H Alternative serialization of the factorized interface **25**

I Inference prompts **27**

I.1 Fine-tuned local models 27

I.2 Prompted API baselines 27

The appendix supplements the main paper with additional related work, then documents corpus selection and target construction, implementation and evaluation details, additional analyses, target-format studies, and the complete inference prompts.

A Additional Related Work

CAD representations and generative models. CAD generation has been studied under several representations. CSG-based approaches recover Boolean construction trees or assemble shapes from compact primitive programs (Du et al., 2018; Yu et al., 2022). B-rep methods instead model the faces, edges, and geometric entities of the final solid (Jayaraman et al., 2022; Xu et al., 2024; Li et al., 2025b). Construction-history representations retain the ordered modeling process: Fusion 360 Gallery provides human construction sequences (Willis et al., 2021), DeepCAD models sketch-and-extrude commands autoregressively (Wu et al., 2021), and Text2CAD conditions a related sequential representation on natural language (Khan et al., 2024). SkexGen further disentangles topology, geometry, and extrusion variation through separate latent codebooks (Xu et al., 2022). These lines motivate structured CAD representations, but their decompositions primarily concern geometric entities, primitives, or latent construction factors rather than the separation of construction from numerical instantiation studied here.

Language models, feedback, and interactive CAD generation. Language models have been applied to both serialized CAD sequences and executable code. Query2CAD uses a general-purpose LLM for direct natural-language CAD generation (Badagabettu et al., 2024); CAD-Llama and CADmium generate structured parametric or sequential CAD representations (Li et al., 2025a; Govindarajan et al., 2026); and Text-to-CadQuery targets executable Python and provides the corpus inherited in this work (Xie & Ju, 2025). More recent systems introduce chain-of-thought reasoning, geometric rewards, or online reinforcement learning for CAD generation and reconstruction (Guan et al., 2025; Niu et al., 2026; Kolodiazhnyi et al., 2026). A complementary line improves an initial prediction through rendered visual feedback, verification, self-refinement, or interaction with CAD tools (Wang et al., 2025; Alrashedy et al., 2025; Li et al., 2026; Gong et al., 2026; Fan et al., 2026; An et al., 2026). Other iterative systems use geometric or morphological evaluators to guide subsequent revisions (Zhou et al., 2026; Kabisov et al., 2026). Sketch–Solver is orthogonal to these feedback mechanisms: its main intervention changes the representation and prediction order before any external repair loop is introduced.

Editing and evaluation. CAD-Editor performs text-guided modification by locating and infilling regions of an existing CAD sequence (Yuan et al., 2025), while Clarify Before You Draw addresses underspecified requests before synthesis (Yuan et al., 2026). Our editing setting is narrower and construction-preserving: the predicted skeleton is retained and only the numerical parameter block is regenerated. Evaluation has also expanded beyond conventional executability and geometric distance. Text2CAD-Bench introduces human-curated tasks spanning broader geometric and application complexity (Wang et al., 2026), while CADTestBench evaluates instruction satisfaction through executable geometric and topological tests (Mallis et al., 2026). These benchmarks target broader capability and functional correctness; our evaluation instead uses matched data, backbone, and decoding to isolate construction–parameter factorization and reparameterization, followed by a separate benchmark for controlled numerical edits.

B Corpus audit and tier selection

Source data. DeepCAD (Wu et al., 2021) represents each model as a sketch-and-extrude construction sequence stored in JSON. Text2CAD (Khan et al., 2024) pairs these sequences with natural-language descriptions at four levels of detail (abstract, beginner, intermediate, and expert). Text-to-CadQuery (Xie & Ju, 2025) then translates each JSON sequence into a CadQuery program with Gemini 2.0 Flash. We use the intermediate description as the instruction and the translated program as the target. The CadQuery programs are therefore LLM translations, whereas the JSON sequences are the original construction data. This provides an independent reference: we replay each JSON sequence with the Text2CAD OpenCASCADE pipeline and compare the resulting solid with the solid built from the paired CadQuery program.

Audit Chamfer distance. The audit uses its own Chamfer distance, which differs from the evaluation metric in Appendix F. We sample 30,000 surface points from each mesh, translate each point cloud to its centroid, scale it by its maximum distance from the centroid, and report the sum of the

two mean squared nearest-neighbor distances multiplied by 10^3. Because each cloud is divided by a single scalar, this audit CD ignores uniform scale but remains sensitive to changes in aspect ratio.

Findings. Of the 176,016 pairs, all JSON references build, 175,967 CadQuery programs parse, and 157,374 (89.4%) build a solid. On these buildable pairs, the median audit CD is 55.3, and only 37.4% fall below 20 (Table 7). A major cause is a scale convention applied twice: Text2CAD normalizes models with a factor of 0.75, and many translated programs multiply this factor into the sketch-plane coordinates again while leaving the extrusion unchanged. The resulting anisotropic discrepancy, with two axes scaled by 0.75 relative to the third, occurs in 41.5% of buildable pairs and cannot be removed by a single normalizing scalar. Normalizing each axis separately lowers the median CD to 6.5. Selection retains isotropic normalization so that aspect-ratio differences remain measurable.

Table 7: Audit CD between each translated CadQuery program and its DeepCAD JSON reference over the 157,374 buildable pairs (30,000 points, $\mathrm{CD} \times 10^3$). Per-axis normalization is shown only as a diagnostic.

Normalization	p10	p25	p50	p75	p90
Isotropic (used for selection)	0.40	6.09	55.27	227.43	451.21
Per-axis (diagnostic)	0.13	0.16	6.52	30.82	77.96

Selection tiers. The four tiers, Verified-CQ@CD5, Verified-CQ@CD10, Verified-CQ@CD15, and Verified-CQ@CD20, retain pairs with audit CD below 5, 10, 15, and 20, respectively; this isotropic threshold is the only selection criterion. The tiers are nested and retain the official Text2CAD train/validation/test split. Each retained pair is then converted into a factorized target and must pass the offline verification in Sec. 3.3, which removes 0.4–0.5% of the records in each tier. Verification is decided once per UID and shared by all tiers, so tier membership does not vary across runs. Table 8 lists the resulting sizes; all controlled models are evaluated on the 2,926-record Verified-CQ@CD20 validation set.

Table 8: Verified corpus tiers. "Audit pass" counts pairs below the threshold and its share of the 157,374 buildable pairs; the split sizes are after offline verification.

Tier	Audit pass	Share	Train	Val	Test
Verified-CQ@CD5	36,642	23.3%	33,001	1,850	1,627
Verified-CQ@CD10	47,027	29.9%	42,388	2,348	2,092
Verified-CQ@CD15	54,232	34.5%	48,878	2,709	2,424
Verified-CQ@CD20	58,813	37.4%	53,035	2,926	2,620

C Corpus construction

This section describes how each selected source program P becomes the aligned training targets: a cleaned plain-code program, a skeleton S, and a parameter block V. The procedure is deterministic and uses no language model, so the same source program and instruction always yield the same targets.

C.1 Source cleaning

The translated programs end with an export call that writes an STL file to a hard-coded path, and they name the final solid with more than 100 different variable names. Table 9 gives the four normalization rules. Their output is both the plain-code target and the input to parameter induction.

A program that no longer compiles after cleaning keeps its original text and is rejected by the verification gates below.

Table 9: Deterministic source-cleaning rules.

Operation	Transformation
Result binding	The variable passed to the export call, or the last assigned variable if there is none, is renamed to `_result_` at its last assignment, and the export call is removed.
Comments and blank lines	Comments and blank lines are removed with the Python tokenizer, so that # characters inside strings are preserved. Narrative comments written by the annotating model account for 19% of source lines, and blank lines for another 19%.
Unused imports	An import is removed when none of the names it binds is read, such as the uncalled visualization import present in 12.8% of programs.
Dead code after the result	Top-level statements after the last `_result_` binding that do not read `_result_` are removed, so that `_result_` is always the final solid.

C.2 Parameter induction

Algorithm 1 gives the deterministic induction procedure.

Algorithm 1 Induction of the skeleton and parameter block.

Require: cleaned source program P and instruction T
Ensure: skeleton S and executable parameter block V

1: **Extract occurrences.** Select every numeric literal in P that is neither zero nor an integer. Integers, including axis indices, right angles, and repetition counts, remain in the skeleton.
2: **Assign roles and types.** Give each occurrence a role, such as radius, height, or x, and a type in {dimension, position, angle, scale, unknown} (Table 10). Use assignment-variable keywords first (`part_2_radius` → radius), then CadQuery argument position (`.circle` → radius, `.extrude` → height, `.moveTo` → x, y).
   Treat a literal assigned to a variable containing `scale`, or multiplied by at least two distinct other literals, as a global normalization factor of type scale. Apply this override by source position, so `sketch_scale = 0.75` does not retype `length = 0.75`.
3: **Share equal occurrences.** Merge same-type occurrences whose values differ by less than 10^{-6} into one parameter that records all source positions. Name it `<role>_<k>`, with k counted per role in extraction order.
4: **Mark stated parameters.** Mark a parameter stated when its value lies within 0.002, half the source-data quantization step, of a number in T.
5: **Build the ratio forest.** Relate dimension parameters using Eq. (2) in a forest rooted at stated parameters; optionally attach a position parameter to half a dimension. Give no relations to other types (Appendix C.3).
6: **Construct S.** Replace every selected occurrence at its source position by its parameter name. Inline pure aliases assigned exactly once to one parameter, such as `length = length_1`, and drop import and export statements. Preserve an existing `_result_` binding; otherwise rename or bind the final solid as `_result_`.
7: **Construct V.** Write stated and free parameters as literals and derived parameters as expressions over their parent, such as `radius_1 = 1/2 * radius_2`. Emit literals before derived assignments and place every derived parameter after its parent (Appendix C.3).

Within identifier cues, position keywords are matched before dimension keywords, so a name such as `hole_center_x` is treated as a position; the scale rule overrides both. Equality sharing is restricted to occurrences of the same type, ratio relations are induced only among dimensions, and the separate centering rule may attach a position to half a dimension. Table 10 summarizes the corresponding roles and static cues.

C.3 Ratio forest and parameter block

Algorithm 2 gives the ratio step of the induction. Its input is the parameter list after equality sharing, in extraction order, with source values x_k, types, and the set $\mathcal{C}$ of stated parameters. The ratio test $\mathrm{fit}(a, b)$ returns the first r in the ordered list $R = (2, \frac{1}{2}, 3, \frac{1}{3}, 4, \frac{1}{4}, \frac{3}{2}, \frac{2}{3})$ with $|a - rb| \le 10^{-9} \max(|a|, |b|, 10^{-12})$, and none otherwise. Because R is closed under reciprocals, the test is symmetric and the candidate graph is undirected. Stated parameters are marked before any traversal, so they are never reached as children; each remaining connected component is rooted

Table 10: Occurrence types and static cues used for typed equality sharing.

Type	Roles	Argument positions	Name keywords
Dimension	radius, length, width, height, diameter, depth, thickness, distance, spacing	`circle`, `rect`, `ellipse`, `box`, `cylinder`, `sphere`, `extrude`, `cutBlind`, `hole`, `cboreHole`, `fillet`, `chamfer`, `shell`, `offset2D`, `radiusArc`	`radius`, `diameter`, `thickness`, `length`, `width`, `height`, `depth`, `size`, `gap`, `dist`, `spacing`
Position	x, y, z, dx, dy, point, offset	`moveTo`, `lineTo`, `line`, `hLine`, `vLine`, `center`, `threePointArc`, `polyline`, `spline`, `pushPoints`, `translate`, `workplane`	`center_x`, `center_y`, `center_z`, `_cx`, `_cy`, `center`, `origin`, `pos`, `_x`, `_y`, `_z`, `offset`
Angle	angle, start angle, countersink angle	`revolve`, `rotate`, `slot2D`, `polarArray`, `cskHole`	`angle`
Scale	scale	multiplied by at least two distinct other literals	assigned to a variable containing `scale`
Unknown	—	none of the above	none of the above

at its first parameter in extraction order, which stays a free literal. The position rule runs after all components are built and uses the same tolerance.

Algorithm 2 Canonical ratio forest.

Require: parameters $1, \dots, n$ in extraction order with values x_k and types; stated set $\mathcal{C}$
Ensure: each parameter is stated, free, or derived as $x_k = r_k\, x_{p_k}$

$\mathcal{D} \leftarrow$ dimension parameters; $\mathcal{P} \leftarrow$ position parameters
$E \leftarrow \{\{i, j\} \subseteq \mathcal{D} : \mathrm{fit}(x_i, x_j) \neq \text{none}\}$
mark every $k \in \mathcal{C}$ stated and every other $k \notin \mathcal{D}$ free
TRAVERSE($\mathcal{D} \cap \mathcal{C}$)
for $k \in \mathcal{D}$ in extraction order, not yet marked **do**
    mark k free; TRAVERSE((k))
end for
for $i \in \mathcal{P}$ marked free **do**
    if some $j \in \mathcal{D}$ satisfies $|x_i - \frac{1}{2} x_j| \leq 10^{-9} \max(|x_i|, |x_j|, 10^{-12})$ **then**
        mark i derived with parent the first such j and $r_i = \frac{1}{2}$
    end if
end for
procedure TRAVERSE(Q) ▷ breadth-first; Q is a queue in extraction order
    while Q is not empty **do**
        $u \leftarrow$ pop the front of Q
        for v adjacent to u in E, in extraction order, not yet marked **do**
            mark v derived with parent u and $r_v = \mathrm{fit}(x_v, x_u)$; append v to Q
        end for
    end while
end procedure

Parameter block. Stated and free parameters are written as literals, using the shortest decimal string that reproduces the source value. A derived parameter is written as `r * parent` for an integer ratio and `p/q * parent` otherwise; its sites in the skeleton still read its own name, so only its right-hand side changes. The block is emitted in levels: all literal assignments first, in the order in which the skeleton first reads their names, followed by derived parameters by depth in the forest, in the same order within each level. Every parent therefore precedes its children. For the numerical reconstruction gate, each literal is parsed as an exact rational, such as $0.0714 = 357/5000$, and the block is evaluated in this order.

Statistics. On the Verified-CQ@CD20 training split, the 53,035 parameter blocks contain 397,531 parameters, of which 12,328 (3.10%) are derived; 10,108 programs (19.1%) contain at least one derived parameter. The ratio $\frac{1}{2}$ accounts for 9,159 derived parameters, 3,511 of them from the position rule, followed by 2 (1,088), $\frac{1}{3}$ (495), $\frac{1}{4}$ (444), $\frac{2}{3}$ (397), $\frac{3}{2}$ (325), 3 (268), and 4 (152). The trees are shallow: 11,995 derived parameters have a root as parent, 329 lie at depth two, and 4 at depth three.

C.4 Verification gates

A target enters the corpus only if it passes the six gates in Table 11, applied in order.

Table 11: Sequential verification gates for induced targets.

Gate	Acceptance condition
Induction	The source parses and yields at least one parameter.
Numerical reconstruction	Solving V in exact rational arithmetic, with free parameters bound to their source values, reproduces every source parameter within an absolute tolerance of 10^{-9}.
Structural validity	Parameter names are unique; every expression refers only to defined parameters and not to itself; the skeleton parses and binds `_result_`; and every parameter can be resolved.
Compilation	The composed program compiles.
Name binding	Every free name read by the skeleton is assigned in V, and `_result_` is bound exactly once.
Geometric agreement	The composed program and cleaned source both execute, and the relative differences of their volumes and surface areas are at most 10^{-5}.

The last gate compares volume and surface area rather than exported files because OpenCASCADE export is not byte-deterministic, whereas these quantities differ by less than 10^{-11} across repeated builds. Its verdict is computed once per UID, stored, and looked up by every tier, so tier membership does not depend on the run. Of the 58,813 pairs selected at Verified-CQ@CD20, 58,812 reach the geometric gate and 58,581 (99.6%) pass it; of the 231 rejected at that gate, 219 composed programs fail to build, 11 build a different solid, and 1 source fails to build.

C.5 Properties of the induced targets

Let S contain reads at sites i and j, and let $\eta(i) = \eta(j) = \theta$. For any assignment environment ρ in which θ is defined, evaluation at both sites returns $\rho(\theta)$. Equality therefore holds for every assignment block that binds the shared name. The learned problem is to generate the correct shared references and binding. This simple property is the reason equality belongs to the skeleton rather than to a post-hoc test of two independently emitted values.

Listing 1: Illustrative induced targets in which selected numerical values become named parameters.

```
# Sketch output S (construction stage)
outer = cq.Workplane("XY").circle(radius_outer).extrude(height)
inner = cq.Workplane("XY").circle(radius_inner).extrude(height)
inner = inner.translate((offset_x, offset_x, 0))
_result_ = outer.cut(inner)

# Solver output V (parameterization stage; composed before S)
radius_outer = 0.50
height = 0.20
offset_x = 0.05
radius_inner = 1/2 * radius_outer
```

The parameter block has three further properties. First, the relations form a forest, so V is acyclic and has a unique solution obtained by evaluating it in order. Second, the targets are a function of (P, T) alone, because every choice in the traversal follows a fixed order. Third, a stated parameter is never derived: stated parameters seed the traversal as roots, and the final position rule applies only to parameters that are still roots and are not stated. A value written in the instruction is therefore never replaced by a relation.

D TWO-STAGE HARNESS AND POST-PROCESSING

D.1 HAND-OFF

Both stages generate Python: Sketch generates the skeleton, and Solver generates the assignment block. The hand-off formatter $u(T, S)$ gives Solver the instruction, the line `# Skeleton`, the complete skeleton, the line `# Parameters to fill (in order)`, and the unresolved parameter names of the skeleton separated by commas (Appendix I). Unresolved names are the names the skeleton reads but never binds, excluding Python built-ins and `math` functions, listed in order of first occurrence. The same routine formats the context during corpus construction and at inference, so Solver sees contexts of identical format in training and deployment.

D.2 INFERENCE AND COMPOSITION

Algorithm 3 specifies deterministic recovery and composition at inference.

Algorithm 3 Inference recovery and program composition.

Require: instruction T and Sketch–Solver generators
Ensure: executable program prediction $\hat{P}$

Generate $\hat{S}$ with Sketch; if present, extract the first fenced code block.
if $\hat{S}$ does not parse, for example because generation was truncated **then**
    obtain unresolved names from the longest parsable prefix of lines, but pass the complete skeleton text to Solver unchanged
else
    obtain unresolved names from $\hat{S}$
end if
Generate $\hat{V}$ with Solver from $u(T, \hat{S})$; strip code fences in the same way.
if $\hat{S}$ does not bind `_result_` **then**
    append a binding to the variable named in an export call; otherwise use the last assigned variable; otherwise use a final bare expression
end if
Bind every unresolved name of $\hat{S}$ not assigned by $\hat{V}$ to `0.0`.
Compose $\hat{P}$ as the import preamble, $\hat{V}$, the fallback bindings, and $\hat{S}$, in that order.
Execute $\hat{P}$ in a subprocess with a 10-second timeout that exports `_result_`.

The zero fallback is a scoring convention for the factorized interface, whose skeleton declares a finite set of required names. It supplies no learned or reference value: an omitted assignment becomes an explicit zero-valued prediction, and the resulting program is charged through IR, CD, and FR_{100}. Direct has no separately declared parameter interface on which the same completion is defined. The rule is shared by all factorized runs; none of the offline verification in Appendix C.4 is used at inference.

D.3 POST-PROCESSING FOR DIRECT GENERATION

Direct outputs, from both fine-tuned and API models, are processed by extracting the first fenced code block if present and appending a `_result_` binding by the same rule only when none exists. Because the plain-code targets already end with `_result_`, an existing binding is never renamed. This append-only rule leaves all 1,500 checked Direct validation targets unchanged; the Sketch–Solver post-processor likewise returns all 2,926 validation targets unchanged.

D.4 TRAIN–INFERENCE SHIFT

Solver trains on induced reference skeletons but receives predicted skeletons at evaluation. Reusing the same formatter prevents a serialization mismatch; it does not remove this exposure shift. In edit mode, the cached Sketch output comes from the system's initial unedited generation and is addressed by the source UID. A missing or malformed cached skeleton remains a model failure.

E TRAINING AND INFERENCE CONFIGURATIONS

E.1 TRAINING

All verified-corpus models share one recipe across the three backbones and across Direct, Sketch, and Solver: full-parameter supervised fine-tuning in bfloat16 for two epochs, learning rate 10^{-5} with cosine decay and 3% linear warmup, per-device batch size 2 with 8 gradient-accumulation steps (effective batch size 16), 8-bit AdamW, gradient clipping at norm 1.0, gradient checkpointing, and loss on assistant tokens only. Sketch and Solver are trained as separate models from the same base checkpoint, each on its own aligned targets. The original-split models use the same full-parameter setup and change only the number of epochs (3), the learning rate (5×10^{-5}), and the gradient-accumulation steps (6, effective batch size 12).

E.2 DECODING AND SEQUENCE LENGTHS

The inference caps are 2,048 tokens for Direct, 2,048 for Sketch, and 1,792 for Solver. Joint symbolic generation uses 3,072. Without shared-name equality, the Solver target is approximately 2.5 times longer and uses a 4,096-token inference cap. Training maximum lengths are 1,536 for Direct, 2,048 for Sketch, 3,072 for the ordinary Solver target, 3,072 for joint generation, and 4,096 for equality removal.

E.3 API EVALUATION SETUP

All API anchors use the same system prompt, four demonstrations, temperature zero, disabled reasoning, and the Direct postprocessor. DeepSeek-v4-Flash uses a 4,096-token cap; DeepSeek-v4-Pro, GPT-5.6-Luna, and GPT-5.6-Terra use 8,192. Table 2 reports their results, and Appendix I reproduces the shared prompt and demonstrations.

F EVALUATION METRICS

Record status and denominators. A record whose reference program does not build is excluded from every denominator; on the original Text-to-CadQuery split, 5,472 of the 5,514 test records have buildable references. For the remaining records, a prediction is *invalid* if the composed program fails to execute within the 10-second budget, does not export a solid, or yields a mesh that cannot be loaded for measurement. Otherwise it is *valid*.

Invalid rate (IR). IR is the percentage of records whose prediction is invalid.

Chamfer distance (CD). For each valid prediction, we load the reference and predicted meshes, translate each by its own mesh centroid, and scale each by the reciprocal of its maximum bounding-box extent. We deterministically sample 10,000 surface points from each mesh. CD is the sum of the two mean squared nearest-neighbor distances, multiplied by 10^3, and is averaged over valid predictions.

Severe-failure rate (FR_{100}). FR_{100} is the percentage of all records whose prediction is invalid or has $\mathrm{CD} \geq 100$. Mean CD is computed only over valid predictions, so a system that fails on more records can remove difficult cases from its CD denominator; FR_{100} uses the full denominator and counts execution failures and severely inaccurate solids together.

Common-valid CD for editing. In the editing experiment, mean CD is computed on the intersection of tasks for which all three routes produce valid predictions within a backbone, so every route is averaged over the same tasks. In Fig. 2, the intersection is taken separately at each edit level. IR and FR_{100} use all 3,366 tasks.

G ADDITIONAL RESULTS AND DIAGNOSTICS

This section complements the aggregate results from three perspectives. We first characterize synthesis by part difficulty and failure type, then examine repeated-value preservation and the representation interventions, and finally analyze parameter-only editing across the number and magnitude of changed values.

G.1 SYNTHESIS BY PART DIFFICULTY

Table 12 stratifies the tier comparison of Table 3 by part difficulty. The difficulty score is a weighted sum of percentile ranks over four program features: the number of numerical parameters (weight 0.55), solid operations (extrude, revolve, loft, and sweep; 0.20), Boolean operations (0.15), and curve operations (0.10). Tertile boundaries are computed once over all parseable source programs, so the 2,926 validation parts are not evenly distributed across tertiles. Every stratum contains the same parts for all rows, and metrics follow Table 3.

Sketch–Solver lowers IR, mean CD, and FR_{100} on medium parts in all four tiers. On easy parts, both formulations have IR below 0.25%, and Sketch–Solver lowers mean CD and FR_{100} in every tier. On hard parts, it lowers all three metrics in Verified-CQ@CD5 and Verified-CQ@CD15, while mean CD is higher in Verified-CQ@CD10 and IR and FR_{100} are higher in Verified-CQ@CD20.

Table 12: Difficulty-stratified comparison across nested corpus tiers with Qwen3-4B. Each metric is reported as Direct / Sketch–Solver; bold marks the better result within each pair.

Tier	Difficulty	n	IR↓	Mean CD↓	FR_{100} ↓
Verified-CQ@CD5	easy	1,390	**0.14**/0.22	5.07/**4.84**	1.94/**1.80**
	medium	915	3.83/**2.30**	14.72/**12.13**	8.42/**5.03**
	hard	621	12.56/**10.47**	23.95/**23.49**	20.77/**16.91**
Verified-CQ@CD10	easy	1,390	0.22/**0.14**	4.33/**3.50**	1.73/**1.01**
	medium	915	3.93/**2.19**	12.97/**10.95**	7.87/**5.03**
	hard	621	10.63/**8.53**	**21.87**/22.82	17.71/**15.46**
Verified-CQ@CD15	easy	1,390	**0.07**/**0.07**	3.30/**2.82**	1.22/**1.01**
	medium	915	3.06/**2.08**	13.05/**11.27**	7.10/**5.25**
	hard	621	11.11/**9.02**	23.86/**21.32**	18.84/**15.78**
Verified-CQ@CD20	easy	1,390	0.22/**0.14**	2.99/**2.69**	1.37/**0.94**
	medium	915	3.06/**2.08**	11.11/**10.96**	6.23/**5.14**
	hard	621	**10.14**/11.76	23.20/**22.49**	**17.87**/18.52

G.2 FAILURE DECOMPOSITION

We analyze the inputs counted by FR_{100} for the Qwen3-4B and Qwen3.5-2B pairs on both evaluation sets: the 5,472 test records with buildable references on the original Text-to-CadQuery split (Table 1) and the 2,926 verified validation inputs (Table 2). Table 13 assigns each failure to one category. Invalid programs are separated into outputs that reach the generation-length limit, programs rejected by the geometry kernel because a three-point arc is degenerate, other kernel failures, and remaining execution errors. Valid programs with $\mathrm{CD} \geq 100$ are separated by whether their feature signature matches the reference. The signature is the multiset of feature operations (extrusion, revolution, Boolean operations, fillets, chamfers, holes, and primitive solids) together with the set of profile primitive types (lines, arcs, circles, rectangles, polygons, and splines).

Length-limit failures. Length-limit failures arise from repetitive continuation rather than long references. For Qwen3-4B on the verified set, no corresponding reference exceeds the limit, and median reference length is 698 tokens for Direct and 572 for Sketch. Every capped output is invalid: Direct repeats nearly identical coordinate calls, while Sketch repeats freshly numbered parameter names without binding the final solid.

Kernel failures. The largest change among kernel failures is in degenerate three-point arcs, which arise when coordinate values make the three arc points collinear or coincident. On the verified set,

Table 13: Decomposition of the failures counted by FR_{100}. Each entry reports the number of inputs as Direct / Sketch–Solver; the last row reproduces the corresponding FR_{100} values from Tables 1 and 2.

		Original split (5,472)		Verified set (2,926)	
	Category	Qwen3-4B	Qwen3.5-2B	Qwen3-4B	Qwen3.5-2B
Invalid	Length limit reached	7 / 5	12 / 7	36 / 32	25 / 21
	Degenerate arc	18 / 12	22 / 11	50 / 22	46 / 15
	Other kernel failure	17 / 16	17 / 24	11 / 21	16 / 17
	Other execution error	9 / 4	12 / 16	1 / 1	4 / 2
$\mathrm{CD} \geq 100$	Features differ	89 / 89	81 / 97	46 / 39	37 / 22
	Features match	135 / 125	121 / 133	55 / 45	28 / 33
Total failures		275 / 251	265 / 288	199 / 160	156 / 110
FR_{100} (%)		5.03 / 4.59	4.84 / 5.26	6.80 / 5.47	5.33 / 3.76

Sketch–Solver reduces these failures from 50 to 22 for Qwen3-4B and from 46 to 15 for Qwen3.5-2B, with the same direction on the original split. Other kernel failures, primarily failed Boolean or face-construction operations, remain mixed across backbones and evaluation sets.

Severe valid failures. Between 43% and 60% of valid programs with $\mathrm{CD} \geq 100$ have the correct feature signature. Among these, 79–90% differ from the reference in the number of rotations and translations, compared with 14–17% of all valid predictions with a matching signature. These programs build the expected features in a different orientation or position, identifying placement as the dominant source of the severe-valid tail.

Program complexity. Failures concentrate on longer programs. On the verified set, the advantage of Sketch–Solver is strongest in the longer half of the reference-length distribution; Table 13 further locates the largest systematic reduction in degenerate three-point arcs. Together, these results place the effect on complex predictions in which numerical choices must remain compatible across a longer construction.

G.3 Repeated-value preservation

We measure repeated-value preservation statically on composed programs from the Qwen3-4B Direct and Sketch–Solver models trained on Verified-CQ@CD10 and evaluated on 2,926 inputs. A *site* is a non-integer numeric literal equal to a value, or a read of a variable assigned that value. For a reference value occupying $m \geq 2$ sites and reproduced at least once by the prediction, preservation requires the prediction to place it at m or more sites. This yields 7,360 values for Direct and 7,347 for Sketch–Solver. Span is the character distance between the first and last literal occurrences in the reference; a value written once and then read by name has no span (Table 14).

Table 14: Repeated-value preservation by span in the reference program. n is the number of counted repeated values.

	Direct		Sketch–Solver	
Span (characters)	n	Preserved	n	Preserved
< 60	2,027	88.3%	2,018	**93.8%**
60–160	576	58.3%	572	**74.3%**
160–400	969	48.6%	957	**59.0%**
≥ 400	938	30.9%	965	**44.2%**
Written once, read by name	2,850	72.3%	2,835	**77.6%**
All	7,360	67.2%	7,347	**75.0%**

G.4 Representation-ablation diagnostics

Joint symbolic generation retains S and V but trains one policy on their concatenation. Removing ratio expressions evaluates every induced ratio into its literal value and retrains only Solver, retaining the same skeleton. Removing shared-name equality maps each selected occurrence to a separate parameter and retrains both LLMs; instantiated numerical arguments remain identical, but targets become substantially longer. Expanded relations retains the original skeleton and ratios, adding exact cross-class equality and same-/cross-class sums or differences whose observed hit rate is enriched by more than $5\times$ over shuffled values. This changes explicit-expression density from 3.10% to 20.8%.

The relation diagnostics use the Verified-CQ@CD10 models on the 2,926 validation inputs. A declared relation is an assignment whose right-hand side is an arithmetic expression over another numerical variable; it is identified by the ratio of child to parent value and by the parent value, independently of names. Recall is the fraction of the 684 relations in the reference parameter blocks that the prediction declares. Precision is the fraction of declared relations that match one of the 2,632 relations in the composed reference programs, which also include relations defining skeleton intermediates. Parent-correct satisfaction takes each reference relation $c = r\,p$, finds the predicted value closest to p, and, when that value equals p, checks whether $r\,p$ also appears among the predicted values.

These diagnostics measure value-level recovery of the reference dependencies. Parent-correct satisfaction is 0.731 for Direct, 0.728 for Sketch–Solver, and 0.689 for expanded relations; relation-matching precision/recall is 0.005/0.001, 0.738/0.273, and 0.586/0.284, respectively. An explicit expression and its correctly inlined value are equivalent at a single operating point, so the synthesis ablation supplies the corresponding end-to-end evidence. The matching operates on parsed values and ratios without requiring semantic site alignment or human constraint labels.

G.5 Expanded-relations intervention

The expanded-relations variant in Table 5 tests whether Solver benefits when more numerical dependencies are written explicitly. Its corpus is derived line by line from the Verified-CQ@CD20 Sketch–Solver corpus and cut into the narrower tiers in the same way. Skeleton and plain-code targets are copied byte for byte, so the Sketch model is reused and only Solver is retrained. Within each parameter block, only literal assignments may be rewritten; scale parameters and the original ratio expressions are left unchanged.

Candidate relations. For each parameter block, we enumerate every numerically exact relation among parameter values in three families: equality $a = b$, ratio $a = rb$, and sum $c = a + b$, which also covers complements $a = c - b$. Each family is counted separately within and across parameter types. Because source dimensions are quantized, many such relations occur by coincidence. We therefore compare each family with a null model that shuffles parameter values across samples within each type, preserving the number of parameters and their marginal distribution. Enrichment is the observed hit rate divided by the shuffled hit rate (Table 15).

Table 15: Numerically exact relation candidates on the Verified-CQ@CD20 training split (53,035 samples, 397,531 parameters). Within-type equalities do not occur because equality sharing has already merged them into one parameter.

Family	Scope	Observed	Shuffled	Enrichment	Used
Sum	cross-type	84,775	8,779	9.7×	✓
Sum	within-type	43,786	4,851	9.0×	✓
Equality	cross-type	24,524	4,094	6.0×	✓
Ratio	within-type	45,260	22,553	2.0×	
Ratio	cross-type	27,378	13,584	2.0×	
Equality	within-type	0	6,979	—	

Rewriting rules. A literal assignment is rewritten as an expression only if three conditions hold. First, its relation belongs to a family enriched by at least $5\times$; both ratio families are excluded. Sec-

ond, the expression evaluates to exactly the original floating-point value, so the composed program performs the same arithmetic and CadQuery calls as before and its geometry is unchanged without re-verification; about 11–18% of sum candidates fail this test by one or two units in the last place. Third, after every parameter is expanded into a linear combination of root parameters, no name written in the expression may have a zero coefficient; this rejects detours such as `y_1 = radius_3 + y_2` with `y_2 = scale_1 - radius_3`. Candidates are ranked by family (equality before sum) and by position in the block, and are accepted greedily unless they would create a cycle. The block is then re-emitted in a stable topological order.

Resulting corpus. On the training split, the fraction of parameters written as expressions rises from 3.10% to 20.83%, and the fraction of samples with at least one expression rises from 19.1% to 52.8%. About one third of the added relations are equalities and two thirds are sums. At 6–10× enrichment, the shuffled-rate estimate attributes roughly 10–17% of the added relations to numerical coincidence. Every rewrite remains exact and preserves geometry, while the induced dependency may differ from the underlying design relation. Listing 2 shows one rewritten block.

Listing 2: One parameter block before and after the expanded-relations rewrite (UID 0001/00018267). The skeleton is identical in both corpora.

```
# Sketch--Solver target          # Expanded-relations target
x_2      = 0.3281               x_2      = scale_1 - x_1
x_1      = 0.4219               x_1      = 0.4219
height_1 = 0.1406               height_1 = pos_1
pos_1    = 0.1406               pos_1    = 0.1406
y_1      = 0.7219               y_1      = scale_1 - pos_2
pos_2    = 0.0281               pos_2    = 0.0281
```

G.6 Editing by number and magnitude of changes

The editing benchmark changes $L \in \{1, 2, 3\}$ instruction-grounded target values and compares Direct regeneration, Sketch–Solver regeneration, and Sketch–Solver skeleton reuse. For $L > 1$, selected parameters have pairwise disjoint dependency cones, distinct old values, and no edited value collides with another selected old value. Selection is greedy by coupling strength. Sets are nested within a source part before certification, but the retained level populations are not nested: only 29 parts pass all three levels. Figure 2 therefore compares performance within each certified level rather than tracking one fixed population across levels.

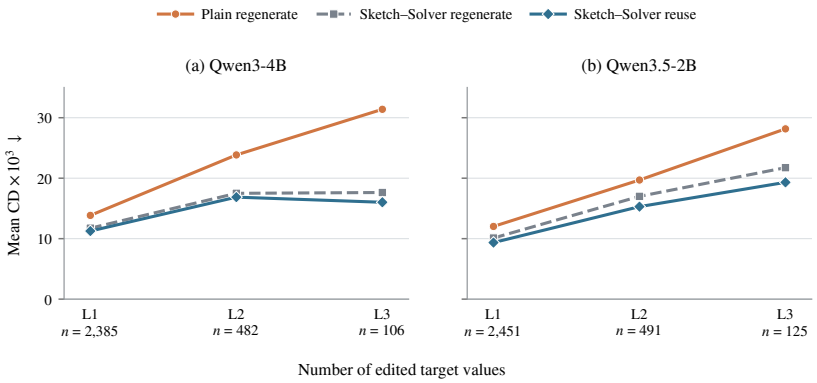


Figure 2: Common-valid mean CD as $L = 1, 2, 3$ target values are edited with Qwen3-4B and Qwen3.5-2B. The three routes are Direct regeneration, Sketch–Solver regeneration, and Sketch–Solver skeleton reuse; n gives the certified task population at each level.

A parameter is an edit candidate if it is a literal in the reference parameter block whose value appears as a number in the instruction, and if its name is read at least twice by the skeleton or referenced by another parameter's expression. For value v and factor k, every number equal to v in the instruction is replaced by kv in its shortest exact decimal form, every independent parameter with value v is set to kv, and the block is re-evaluated so that dependent parameters follow. The edited reference is

composed with the unchanged skeleton by the same merge used at inference, and its mesh is built from the composed program. The observable quantities are the values of all parameters and skeleton intermediates together with the three bounding-box extents of the reference mesh. A task is certified only if it passes every check in Table 16; of 4,127 candidate tasks, 3,366 (81.6%) are certified, and a part is retained at a level only if its tasks pass at all three factors.

Table 16: Certificates for edited reference tasks.

Certificate	Condition
Build	The edited reference builds a solid.
Linearity	Every changed program quantity scales by exactly k, and at least one changes.
No collision	No unchanged quantity equals an edited value v.
No stale value	Apart from v itself, no changed quantity has its old value in the instruction.
Effect	The edited geometry differs from the original.
Not a similarity	The three bounding-box extent ratios are not all equal.

Across edit magnitudes on Qwen3-4B, common-valid mean CD for Direct regeneration / Sketch–Solver regeneration / skeleton reuse is 19.29/17.45/16.69 at $k = 0.5$, 14.83/10.79/10.60 at $k = 1.5$, and 14.12/10.33/9.66 at $k = 2.0$. Contraction is harder for all routes. On Qwen3.5-2B, the same qualitative route ordering holds in the aggregate. Reuse generates a new full assignment block and can change its expression structure; only the cached skeleton and its equality-sharing pattern are held fixed.

G.7 Qualitative parameter-only edits

Figure 3 visualizes eight certified construction-preserving edits selected for consistent performance ordering across both backbones and visually legible geometric changes. The displayed predictions and CD values are from Qwen3-4B. Five examples modify one target value and three modify two values. All panels within a row share one scale and are recentered without rescaling, so the requested dimensional change remains visible; the pale square beneath each part is the shared scale box.

H Alternative serialization of the factorized interface

We compare two serializations of the same factorized targets on the original Text-to-CadQuery split. This auxiliary study isolates how Sketch communicates the parameter inventory to Solver.

In the explicit-inventory serialization, Sketch emits a JSON object that first lists all parameter names and then gives the skeleton as a string, while Solver emits a JSON list of parameter records. In the AST-mediated serialization, Sketch emits the skeleton as Python and its unresolved parameter names are recovered from the predicted code before invoking Solver. Both variants were trained with LoRA on Qwen3-4B from the same labels and decoded greedily on the 5,464 buildable validation records of the original Text-to-CadQuery split.

Table 17: Comparison of explicit-inventory and AST-mediated serializations of the factorized interface.

Sketch output	Parameter names	IR↓	Unclosed name list
JSON: name list, then skeleton string	generated by Sketch	23.01	21.98
Python skeleton	recovered by AST	**2.54**	—

With the explicit-inventory serialization, 1,201 of the 5,464 predictions never close the name list: Sketch keeps emitting new parameter names (a median of 236) until the token budget is exhausted, so no skeleton is produced. These account for almost all of its 23.01% IR. Recovering names from the skeleton removes this failure mode and lowers IR to 2.54%. Sketch–Solver therefore uses the AST-mediated hand-off and represents the Solver output as Python assignments, so both stages emit Python and composition reduces to concatenation (Appendix D.2).

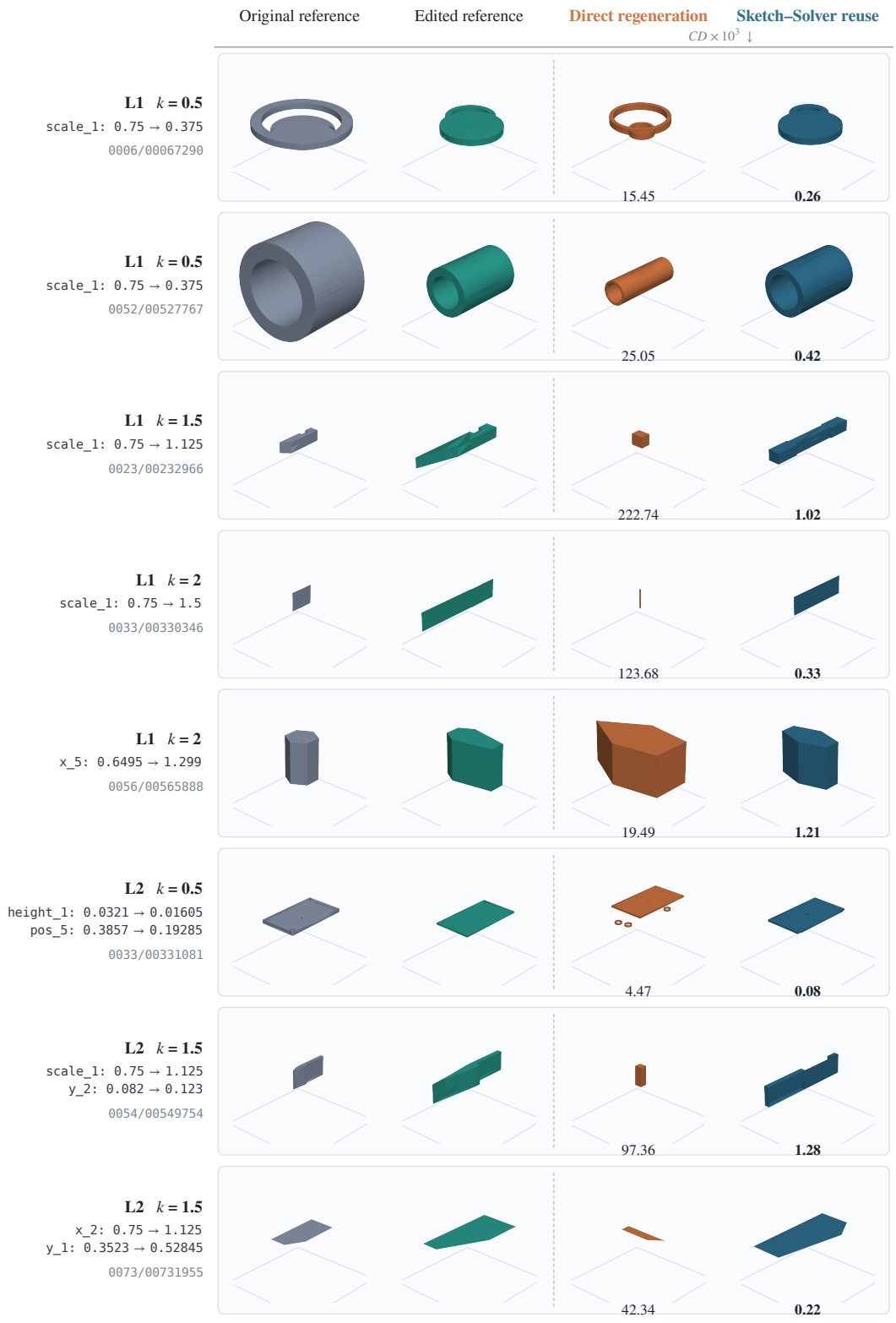


Figure 3: **Qualitative parameter-only edits with Qwen3-4B.** Columns show the original reference, the edited reference, Direct regeneration, and Sketch–Solver skeleton reuse; numbers under the two prediction panels are CD $\times 10^3$ against the edited reference, lower in bold. Rows 1–5 are L1 edits and rows 6–8 are L2 edits. The left margin gives the edit level, magnitude k, edited parameters with their old and new values, and source UID.

I INFERENCE PROMPTS

This section records the message content used at inference. The local fine-tuned models use the same system prompts during training and evaluation. For every backbone, messages are rendered by its native chat template with reasoning disabled. Angle-bracketed fields below denote sample-specific content inserted by deterministic code.

I.1 FINE-TUNED LOCAL MODELS

Direct baseline. The Direct model receives one system message followed by the natural-language instruction as the user message:

```
[system]
You are a CAD expert. Convert the natural language instruction into CadQuery code. Output ONLY
      the Python code, ending with `_result_ = ...`.

[user]
<INSTRUCTION>
```

The assistant generates the complete CadQuery program in one response.

Sketch stage. Sketch receives the instruction unchanged:

```
[system]
You are a CAD expert. Convert the natural language instruction into a parameterized CadQuery
     program SKELETON. Replace parameterizable design values with symbolic parameter names (e.
     g. length_1, radius_1, pos_1). Keep structural constants such as zeros, axis indices,
     right angles, and repetition counts unchanged. Output ONLY the Python code, ending with `
     _result_ = ...`. Do NOT output JSON or prose.

[user]
<INSTRUCTION>
```

Solver stage. Solver receives the same instruction, the complete predicted skeleton, and the parameter names extracted from that skeleton in first-occurrence order:

```
[system]
You are a CAD expert. You are given a natural language instruction and a parameterized
     CadQuery skeleton. Output ONLY a block of Python assignments giving every symbolic
     parameter its value, one per line, in an order where each name is defined before it is
     used:
  length_1 = 0.75
  radius_1 = 1/2 * length_1
Write a literal number for each independent parameter. When a parameter is represented through
      another parameter, write a supported arithmetic expression such as a simple ratio.
     Output no prose, no comments, no code fences, and nothing but the assignments.

[user]
<INSTRUCTION>

# Skeleton
<PREDICTED SKELETON>

# Parameters to fill (in order)
<COMMA-SEPARATED PARAMETER NAMES>
```

For parameter-only editing, the message format and system prompt are unchanged: `<INSTRUCTION>` is replaced by the revised instruction and `<PREDICTED SKELETON>` by the cached skeleton from the original generation.

I.2 PROMPTED API BASELINES

All four API models receive the same system message, the same four alternating user/assistant demonstration pairs, and the evaluation instruction as the final user message. The demonstrations below are copied from the training split and are disjoint from the evaluation IDs. Source UIDs are reported for provenance but are not included in the messages.

System prompt for API inference.

```
You are a CAD expert. Convert the natural language instruction into a CadQuery program.

Rules:
1. NO COMMENTS. Not a single `#` line, no inline `#` after a statement, no docstrings. Do not
     reason, plan, annotate or narrate inside the program: no "# Part 1", no "# approximate",
     no "# we'll do X". Work it out silently and emit only the finished statements. A program
     with comments is rejected.
2. Output ONLY Python code | no explanation before or after it. If you use a code fence, use
     exactly one and close it.
3. Begin with `import cadquery as cq`. If your program calls any `math.*` function, you must
     also `import math`. Import nothing else.
4. Bind the final solid to `_result_`. That binding must appear exactly once, as the last
     statement of the program. `_result_` is what gets exported, so a program that builds the
     right shape but never binds it produces nothing.
5. Do not call `exporters.export(...)`, do not write files, do not call `show_object(...)`.
6. Use the numbers the instruction gives, verbatim and in the units it uses. Parts in this
     dataset are normalised so the largest bounding-box extent is 0.75.
7. `rect(...)` and `circle(...)` are centred on the workplane origin; move geometry with `.
     translate((x, y, z))` or by offsetting sketch points.
8. Write straight-line modelling code: no functions, no classes, no loops, no conditionals.
9. Use only real CadQuery API. If you are unsure a method exists, use one you are sure of.
10. Keep the program short. These parts need a handful of statements, not dozens.
```

Demonstration 1 (source UID 0007/00073096).

```
[user]
Start by creating a cylindrical base. The base has a circular cross-section and is extruded to
      form a solid body. The height of the cylinder is 0.075 units, while both the length and
     width are 0.75 units. The cylinder is symmetrical and forms the foundation of the model.

[assistant]
import cadquery as cq
part_1_radius = 0.375 * 0.75
part_1_height = 0.075
part_1 = cq.Workplane("XY").circle(part_1_radius).extrude(part_1_height)
_result_ = part_1
```

Demonstration 2 (source UID 0039/00398834).

```
[user]
The design involves creating a cylindrical object with a central circular hole. Start by
     setting up a new coordinate system with default angles and vectors. Draw two concentric
     circles on a single face, with the outer circle defining the cylinder's outer boundary
     and the inner circle forming the central hole. After scaling the entire sketch to fit the
      desired dimensions, transform it into a 3D model. Finally, extrude the sketch to create
     a solid cylinder with a height of 0.2107 units, a length and width of 0.75 units, and a
     central hole that runs through the entire length of the cylinder.

[assistant]
import cadquery as cq
outer_radius = 0.375 * 0.75
inner_radius = 0.1944 * 0.75
height = 0.2107
wp = cq.Workplane("XY").circle(outer_radius).extrude(height)
_result_ = wp.cut(cq.Workplane("XY").circle(inner_radius).extrude(height))
```

Demonstration 3 (source UID 0068/00682254).

```
[user]
Create a rectangular block with two circular holes. The block has a length of 0.75 units, a
     width of 0.3 units, and a height of 0.125 units. The first face features a rectangular
     area with a circular hole in the center. Similarly, the second face has another
     rectangular area with a circular hole in the center. These two faces are aligned along
     the length of the block, forming a symmetrical structure. The block is created by
     extruding the combined shape of these two faces along the height dimension.

[assistant]
import cadquery as cq
length = 0.75 * 0.75
width = 0.3 * 0.75
height = 0.125
hole_radius = 0.0625 * 0.75
hole_center_1 = (0.125 * 0.75 - length/2, 0.15 * 0.75 - width/2)
hole_center_2 = (0.625 * 0.75 - length/2, 0.15 * 0.75 - width/2)
rect = cq.Workplane("XY").rect(length, width).extrude(height)
_result_ = rect.faces(">Z").workplane().pushPoints([hole_center_1, hole_center_2]).hole(2 *
     hole_radius)
```

Demonstration 4 (source UID 0097/00975607).

```
[user]
The first part of the multi-part CAD model consists of a rectangular prism and a pentagonal
     prism. The rectangular prism has a length of approximately 0.086 units, a width of 0.1118
      units, and a height of 0.5041 units. The pentagonal prism, after being scaled down by
     0.75, has a base that forms a pentagon and a height of 0.5041 units. Both prisms are
     aligned along the same axis and are combined to form the final shape. The overall
     dimensions of the first part are a length of 0.7500000052762805 units, a width of
     0.34907559035371855 units, and a height of 0.5041313162650018 units.

[assistant]
import cadquery as cq
part_1_width = 0.086 * 0.75
part_1_height = 0.1118 * 0.75
part_1_depth = 0.5041
part_1 = cq.Workplane("XY").rect(part_1_width, part_1_height).extrude(part_1_depth)
part_2_points = [
    (0.3127 * 0.75, 0.0852 * 0.75),
    (0.3127 * 0.75, 0.3491 * 0.75),
    (0.5517 * 0.75, 0.3491 * 0.75),
    (0.75 * 0.75, 0.1136 * 0.75),
    (0.4874 * 0.75, 0.1136 * 0.75)
]
part_2 = cq.Workplane("XY").polyline(part_2_points).close().extrude(part_1_depth)
_result_ = part_1.union(part_2)
```

After these four pairs, the evaluation instruction is appended as the final user message.