METAGEN: A DSL, DATABASE, AND BENCHMARK FOR VLM-ASSISTED METAMATERIAL GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Metamaterials are micro-architected structures whose geometry imparts highly tunable—often counter-intuitive—bulk properties. Yet their design is difficult because of geometric complexity and a non-trivial mapping from architecture to behaviour. We address these challenges with three complementary contributions. **(i)** ***MetaDSL***: a compact, semantically rich domain-specific language that captures diverse metamaterial designs in a form that is both human-readable and machine-parsable. **(ii)** ***MetaDB***: a curated repository of more than $150\,000$ parameterized MetaDSL programs together with their derivatives—three-dimensional geometry, multi-view renderings, and simulated elastic properties. **(iii)** ***MetaBench***: benchmark suites that test three core capabilities of vision–language metamaterial assistants—structure reconstruction, property-driven inverse design, and performance prediction. We establish baselines by fine-tuning state-of-the-art vision–language models and deploy an omni-model within an interactive, CAD-like interface. Case studies show that our framework provides a strong first step toward integrated design and understanding of structure–representation–property relationships.

1 INTRODUCTION

Metamaterials represent a key frontier in materials science: by exploiting small, patterned geometries, they endow bulk materials with properties beyond those of the constituent substance. Careful geometric tuning yields extraordinary behaviours such as programmable deformation (Jenett et al., 2020; Babaee et al., 2013), extreme strength-to-weight ratios (Qin et al., 2017), and materials that are both stiff and stretchy (Surjadi et al., 2025). The design space is effectively limitless, with exciting applications ranging from thermal management (Fan et al., 2022; Attarzadeh et al., 2022) to biomedical implants (Ataee et al., 2018; Ambu & Morabito, 2019).

Despite this promise, neither the design nor the downstream adoption of metamaterials have realized their full potential. This is largely due to three long-standing hurdles: (i) navigating the immense geometric diversity of candidate architectures; (ii) characterizing the intricate structure–property relationship; and (iii) collating information and assets from the highly-fragmented literature base, which spans several fields and contains considerable variation in terminology, assumptions, geometry descriptors, and evaluation protocols (Makatura et al., 2023; Lee et al., 2024; Xue et al., 2025). These hurdles create a consistently high barrier to entry, whether your goal is to generate a new structure or simply identify an existing one that is suitable for some application.

Vision–language models (VLMs) are poised to address this, as they excel at the cross-modal reasoning, retrieval, and generation required for effective metamaterial design – spanning text, images, 3-D geometry, and numerical property vectors. VLMs could also democratize metamaterial design by exposing a unified knowledge base via natural-language, complete with iterative conversational formats that foster human-in-the-loop material design over a shared context. Unfortunately, high-quality data curation presents a significant barrier for VLM training and more general data-driven metamaterial design approaches, due to the three hurdles discussed above (Lee et al., 2024).

To address this issue, we introduce a general, extensible ecosystem for AI-assisted metamaterial design, anchored by 3 components:

1. **MetaDSL**: a domain-specific language that captures metamaterials in a structured, compact, and expressive form accessible to both humans and large language models.

2. **MetaDB**: a database of more than 150 000 metamaterials, each of which pairs a MetaDSL program with the derived 3-D geometry, rendered images, and simulated properties.
3. **MetaBench**: benchmark suites that probe three fundamental metamaterial design tasks – structure reconstruction, property-driven inverse design, and performance prediction – using data sampled from MetaDB.

To complete our vision, we use MetaBench to train and evaluate *MetaAssist*, a VLM assistant baseline and interactive CAD environment that facilitates multi-modal design interactions including language, images, geometry, and MetaDSL code.

All four components are designed for extensibility and community contribution, such that they can evolve seamlessly alongside the state of the art in materials science and agentic design. Collectively, our ecosystem provides a coherent, extensible knowledge base for metamaterial design, while laying the foundation for intuitive, efficient human–AI collaboration in architected materials.

2 Background

Metamaterials Experts commonly use forward design to craft parameterized structures for specific targets (Muhammad & Lim, 2021; Frenzel et al., 2017; Meier et al., 2025). Inverse design approaches (Lee et al., 2024) are often driven by data-informed search over particular shape representation sweeps. Panetta et al. (2015) analysed 1205 families of cubic truss lattices, while Abu-Mualla & Huang (2024) expanded to 17 000 truss structures spanning six crystal lattices. High-throughput workflows also consider thousands of thin-shell architectures including plate lattices (Sun et al., 2023a) and TPMS-inspired surfaces (Xu et al., 2023; Liu et al., 2022; Yang & Buehler, 2022; Chan et al., 2020). Because many datasets target a single architecture class (e.g. beams or shells) and a narrow performance metric, they restrict the attainable property gamut and thus the capability of downstream models (Berger et al., 2017; Lee et al., 2024). Recent designs also increasingly blend classes in hybrid or hierarchical forms (Surjadi et al., 2025; Chen et al., 2019; White et al., 2021), emphasizing the need for representations that span such boundaries. The procedural-graph approach of Makatura et al. (2023) captures diverse geometries but is demonstrated primarily for human-in-the-loop workflows. Voxel and hybrid encodings scale to 140 k–180 k diverse structures (Yang et al., 2024a; Xue et al., 2025), but they sacrifice semantic clarity and compactness, which complicates human or agent editing. Such tradeoffs – along with inconsistencies in geometry descriptors, vocabularies, and evaluation protocols – continue to impede dataset reuse and extensibility (Lee et al., 2024). We close these gaps with a universal metamaterial descriptor (MetaDSL) along with a reconfigurable database of 150 000 metamaterials (MetaDB). Each MetaDB entry couples a succinct, semantically rich program with derived 3-D geometry, renderings, and simulated properties, enabling consistent comparison and seamless expansion. Programmatic templating further enlarges the design space, and community contributions can grow both MetaDB and the accompanying benchmark suite.

Vision–Language Models for Design Vision–language models (VLMs) have permeated design tasks, including procedural textures (Li et al., 2025), 3-D scenes (Yang et al., 2024b; Kumaran et al., 2023), mesh generation and editing (Sun et al., 2023b; Wang et al., 2024; Jones et al., 2025; Huang et al., 2024; Yamada et al., 2024), interior layouts (Çelen et al., 2024), sewing-pattern synthesis (Nakayama et al., 2025; Bian et al., 2025), and computer-aided engineering and manufacturing (Makatura et al., 2024a;b; Choi et al., 2025; Yuan et al., 2024). In most cases, code serves as the medium: pretrained models follow instructions, reuse standard patterns, and emit domain-specific scripts (e.g. Blender Python). When tasks demand novel grammars or specialist knowledge, fine-tuning further elevates performance (Zhou et al., 2025). Our work adopts this code-centric philosophy but tailors it to metamaterials, whose design demands rich geometric semantics, physical constraints, and fluid translation among text, images, programs, and numerical property vectors. By grounding the interface in a purpose-built DSL and a physically validated database, we lay a robust foundation for future VLMs to reason about, generate, and refine architected materials at scale.

3 Domain-Specific Language

Metamaterial design hinges on precise, expressive geometry representation, but the representational demands (RD) increase dramatically in service of a unified, extensible ecosystem. To be usable

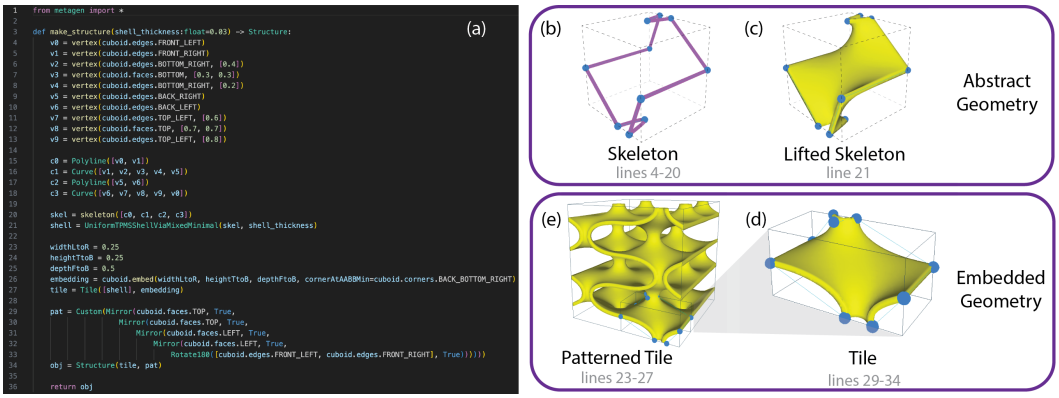


Figure 1: A MetaDSL program (a) and illustrations of each construction stage (clockwise): **(b)** build a 1D skeleton relative to an abstract convex polytope Π_{abs} – here, a cuboid; **(c)** specify a lifting procedure from 1D to 3D; **(d)** embed Π_{abs} in $\mathbb{R}^3$ to create a tile, and execute the lifting procedure to create our final geometry; and finally, **(e)** tessellate the tile according to the specified pattern.

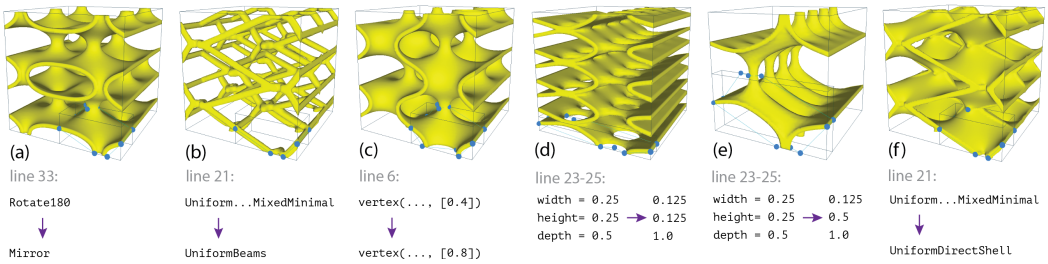


Figure 2: We illustrate the expressive power of MetaDSL by showing six different structures that all stem from the program shown in Figure 1(a). Each one is produced by changing a single aspect of the original program, as detailed below each structure.

and developable by both humans and computational agents, our representation must be: (RD1) expressive enough to support the full range of metamaterial architectures; (RD2) modular and reconfigurable; (RD3) compact, semantically meaningful, and easy to use; (RD4) amenable to and robust under generative design; (RD5) verifiable and valid-by-construction and (RD6) flexible, to support experimentation and extensions. MetaDSL lays out a long-term design philosophy uniquely amenable to these goals. While our current implementation tackles a core subset (Section 3.2), our infrastructure is built extensibly. This will facilitate the evolution of MetaDSL, such that new design paradigms can be added as metamaterial research matures, without invalidating existing programs.

3.1 Language Design Philosophy

MetaDSL uses a modular, compositional approach supported by a rich type system that determines compatibility between components. This promotes flexibility while ensuring verifiable outcomes. As shown in Figure 1, our highest-level decomposition mimics a hierarchical approach that is common in metamaterial design: *tiles* are used to describe small representative units of a structure's geometry, while *patterns* propagate the tiles into a space-filling structure. These levels are independent and polymorphic, such that a pattern can be applied to any number of tiles, and vice-versa.

Tiles Tiles serve two purposes in MetaDSL: (1) representing structural geometry within some finite, embedded convex polytope (CP), $\Pi_{\text{emb}} \subset \mathbb{R}^3$; and (2) maintaining structured information about their contents, which can be queried to determine validity and/or compatibility. To facilitate these goals, tile contents are defined using local coordinates relative to Π_{abs}, which is an abstract (non-embedded) CP (e.g., cuboid, tetrahedron) of the same type as Π_{emb}. For example, the MetaDSL snippet `v0=vertex(cuboid.edges.TOP_LEFT, [0.5])` instantiates a vertex of type `PointOnCPEdge` at the midway point of the cuboid edge; similarly, an edge between `v0` and

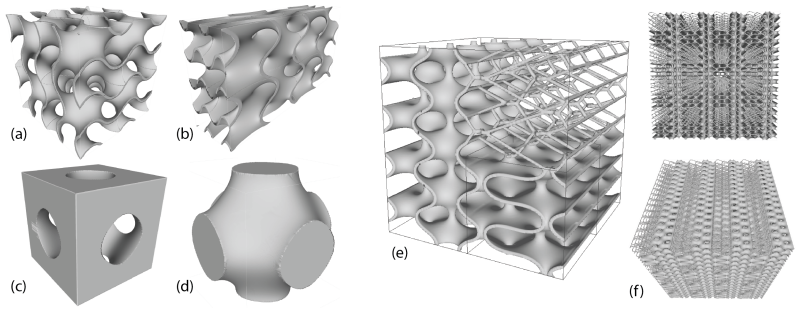


Figure 3: MetaDSL structures that would be impractical or impossible to generate using our default geometry kernel, ProcMeta. (a-d) Structures extracted from implicit functions using our Implicit and CartesianVolume language extensions: (a) Sheet Gyroid TPMS; (b) Sheet Gyroid TPMS with a 1:2:4 aspect ratio, made possible by selecting a non-uniform embedding for the containing Tile; (c) exo-network of the Schwarz P TPMS; (d) endo-network of the Schwarz P. (e,f): An example of a multi-tile pattern, which composes structures from Figure 1(e) and Figure 2(b,c) to form a larger triply-periodic cell; (e) shows a single multi-tile unit, while (f) shows two views of a 4x4x4 block.

`v1=vertex(cuboid.faces.TOP)` would return an `EdgeContainedWithinCPFace`. As typed vertices and edges are combined into larger structures such as a `skeleton`, their classifications are used to deduce and record broader structural relationships. For example, simple queries on the skeleton in Figure 1(b) would reveal that it comprises a single connected component of type `SIMPLE_CLOSED_LOOP`, and it has 1-dimensional incidence on every face of Π_{abs}. This classification is used to judge a skeleton's suitability for a given *lifting function*, which is the procedure used to promote a skeleton into volumetric geometry, or `LiftedSkeleton`. To embed our lifted skeleton, we need only assign a set of corner positions that maps $\Pi_{\text{abs}} \to \Pi_{\text{emb}} \subseteq \mathbb{R}^3$, then instantiate the relative vertices of the lifted skeleton accordingly. The proposed embedding is validated by Π_{abs} to ensure that it preserves convexity and any angle or length constraints specific to Π_{abs}.

There are many approaches to populate a tile using these primitive elements (see Section 3.2). However, the primary contribution is the Tile itself: although many works rely on the *concept* of a tile (Makatura et al., 2023; Panetta et al., 2015; Abu-Mualla & Huang, 2024; Mirramezani et al., 2025), no previous works instantiate tiles as entities against which structural elements can be referenced, analyzed, and queried in the service of verifiable composition (RD5). Our distinction between Π_{abs} and Π_{emb} also improves modularity (RD2) by seamlessly changing tile embeddings, as shown in Figure 2(d,e). Our relative position specifiers also increase semantic meaning and readability (RD3), facilitate robust exploration (RD4), and circumvent the numerical values and computations that often prove challenging for VLMs (RD3, RD4) (Makatura et al., 2024a;b).

Patterns To promote a tile into a space-filling object, MetaDSL applies a *pattern* composed of spatial repetition procedures such as mirrors and rotations. Patterns can only be applied to embedded tiles, because the admissible pattern operations are influenced by extrinsic geometric measures such as the dihedral angles between planes of Π_{emb}. However, our pattern operations use lazy evaluation, such that they can be pre-composed over a generic Π_{abs}. As before, each pattern operation is specified using local coordinates relative to Π_{abs}, such as a mirror across `cuboid.faces.TOP`. By composing these patterns, tiles can be propagated according to e.g. periodic tilings given by crystallographic space groups (Adams & Orbanz, 2023), or perhaps even aperiodic tilings given by a procedural pattern generator. The tile's structured information allows us to verify local pattern compatibility based on boundary adjacency, so even complex patterns can yield coherent metamaterials.

The combination of a Tile and a Pattern yields a `Structure`, which is a complete representation of the metamaterial. In a final layer, we provide standard constructive solid geometry (CSG) Boolean operations to combine multiple Structures. This makes it easy to define metamaterials with mixed scales, interpenetrating lattices (White et al., 2021), or multi-tile patterns, as shown in Figure 3(e,f).

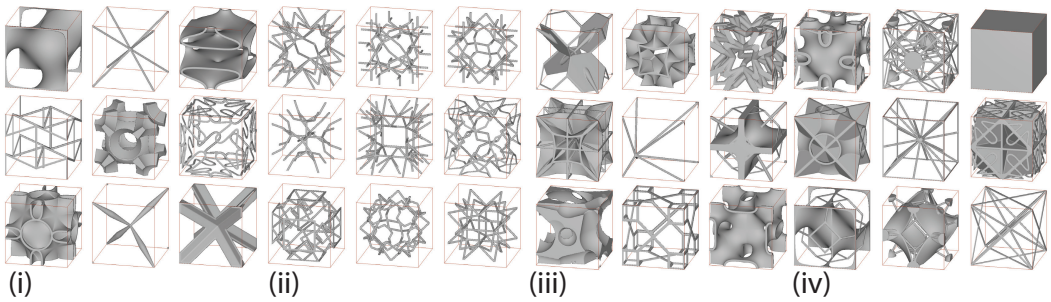


Figure 4: Assortment of metamaterials in MetaDB, illustrating four creation modes: (i) hand-authored seeds, (ii) generated models, (iii) type-enabled mutations, and (iv) LLM-augmented hybrids.

3.2 Implementation

We implement MetaDSL as an embedded DSL in Python, which provides a familiar, flexible interface with support for comments, descriptive identifiers, and programmatic constructs such as loops, modules, and parameterization (RD2, RD3, RD5). MetaDSL does not include a geometry kernel, but our Structure objects can be transpiled to any number of existing representations. We target the graph format of Procedural Metamaterials (ProcMeta) (Makatura et al., 2023), as their skeletal design space is specifically designed to support a variety of metamaterial classes (RD1), and their approach directly inspired the initial set of abstract CPs and lifting functions implemented in MetaDSL. For a detailed description of the current language design, implementation, system design insights, and a comparison to ProcMeta, please see Section C. The complete MetaDSL documentation is in Section I.2.

The limitations imposed by our ProcMeta backend also influenced the core functionality we implemented. For example, MetaDSL currently prioritizes patterns that yield translationally-tileable unit cubes, as ProcMeta only produces geometries in that scope. However, as MetaDSL can be transpiled to any kernel, it is not inherently bound by these limitations. As a proof-of-concept, we extended MetaDSL to support implicit function-based skeleton generators and SDF-based lifting functions. To do this, we introduced a `CartesianVolume` object, which anchors a Cartesian grid to the entities of Π_{abs}, such that local Cartesian coordinates can automatically be transformed into CP-referenced entities. We also wrote a basic geometry kernel and an accompanying translation layer to go from a MetaDSL structure to our kernel input format. As shown in Figure 3(a-d), these constructs allow MetaDSL to capture myriad structures that are common in metamaterial literature (Fisher et al., 2023), yet impractical or impossible to represent in ProcMeta. Our separate geometry kernel also relaxes the limitations of ProcMeta, by allowing e.g. translational units that reside in something other than a unit cube. This highlights the flexibility of MetaDSL's design philosophy and its ability to expand without invalidating existing examples.

4 Database Generation

MetaDSL represents metamaterials in a consistent, concise manner, which permits a single pipeline that produces code, watertight geometry, renderings, and simulated properties for *every* entry. To ensure the quality of MetaDB, we only add *validated* models that pass basic checks (see Section D.6).

4.1 Constructing Metamaterial Models

Each metamaterial is a DSL program, or *model*, that may optionally expose a set of design parameters (with default values). Our metadata block also allows program authors to include details such as bounds, dependencies, or recommended ranges for each parameter. This clarifies design freedom, enables continuous exploration, and provides hooks for optimisation schemes. The metadata is stored in a machine-parsable format (YAML) with a prepopulated scheme for tracking e.g. provenance, versioning, and notable traits about the structure, including symmetries, architecture type (beams, shells, etc.), and related structures. Our metadata also permits custom fields.

Direct Construction *Authored* models are human-written, with provenance records tracking the model author and the original design source(s), and editable semantic parameters to encode families of models. We also provide a programmatic *generator* interface to create families of models. As a proof of concept, we implemented a generator following Panetta et al. (2015); this generates parametrized models for all 1,205 truss topologies using a few hundred lines of Python. Our type-checked DSL allows us to specify and evaluate validity constraints on the small tile, without needing to generate the fully-patterned beam network. Moreover, because our generator is exposed and editable, we can easily modify the high-level generator parameters (e.g. maximum vertex valence) to output different sub- or supersets of interest. For each *generated* model, the provenance metadata stores the generator script, its settings, and per-instance parameters; generator parameters may be substituted for specific values or passed through to remain exposed in the resulting programs.

Augmentation We propose two orthogonal protocols to enlarge MetaDB based on existing models. Our first strategy, *Hybridization* (crossover), is motivated by works that offer unique, extremal mechanical properties by hybridizing common structures such as trusses+woven beams (Surjadi et al., 2025), nested trusses (Boda et al., 2025), TPMS shells+planar shells (Chen et al., 2019), and trusses+solids (White et al., 2021). We emulate this process by prompting an LLM with pairs or triplets of parent programs, then requesting hybrid code. Our prompting strategy (detailed in Section D.3) follows insights from recent works in LLM-mediated program search (Li et al., 2025; Romera-Paredes et al., 2024). The resulting *hybridized* model stores its parent IDs, prompt details, and LLM details as provenance information.

Our second strategy, *mutation*, leverages MetaDSL's type system to apply targeted edits—such as skeleton reconfiguration, pattern adjustment, and lift procedure changes—while guaranteeing validity. The operators are described in Section D.4. These operations are motivated by works such as Akbari et al. (2022), which posits beam approximations of TPMS shells. Each mutation stores its parent and details about the mutator function.

4.2 Auxiliary Data Generation

For every model we generate three auxiliary artifacts: geometry, renderings, and physical property predictions. To obtain the geometry, we transpile our MetaDSL model into a ProcMeta graph (Makatura et al., 2023) and use their geometry kernel to export a watertight `.obj`. Using the exported mesh, our custom PyRender scene produces orthographic images from the front, top, right, and front-top-right viewpoints. Finally, we use the integrated simulations of ProcMeta to voxelize the mesh on a 100^3 grid and perform periodic homogenisation using a base material with $E{=}1$, $\rho{=}1$, $\nu{=}0.45$. The resulting $6{\times}6$ stiffness matrix C is reduced to 18 scalars: six global metrics—Young's modulus E, shear modulus G, Poisson ratio ν, bulk modulus K, anisotropy A, volume fraction V—plus directional values for E (3), G (3), and ν (6). More details are available in Section D.5. MetaDB therefore combines code, geometry, simulation, imagery, and rich provenance—providing a unified benchmark and a data-efficient training ground for vision–language metamaterial assistants.

5 Benchmark Curation

From MetaDB we derive a benchmark that covers three fundamental metamaterial tasks: **(1) reconstruction**—produce a DSL program that reproduces a target structure (for example, from images); **(2) material understanding**—predict the property profile of a given structure description; and **(3) inverse design**—generate a DSL program that satisfies a requested property profile. Each task supports multiple *query types* based on the inputs available. For instance, material understanding may be invoked with a single image ("1-view") or with four images plus code ("multiview_and_code"). The benchmark suite ships a dataset for every query type.

5.1 Task-Based Dataset Construction

We start with a designated pool of *active* models and partition them into train, validation, and test splits that remain fixed for all tasks. The relevant information for each query type is as follows.

Table 1: MetaAssist baselines evaluated on MetaBench. LLaVABase is not reported because it failed to produce any valid output. See Figures 15 to 17 for qualitative evaluation.

Category	Inverse Design		Material Understanding	Reconstruction		
Metric	Error ↓	Valid ↑	Error ↓	CD ↓	IoU ↑	Valid ↑
Model						
LLaVA	.021 ± .002	**98.1%**	.030 ± .004	**.032 ± .001**	**.493 ± .008**	94.2%
Nova	**.018 ± .002**	89.5%	**.021 ± .003**	.035 ± .001	.449 ± .007	**97.5%**
NovaBase	.056 ± .023	2.6%	.199 ± .005	.119 ± .003	.051 ± .003	19.3%
OpenAIO3	.028 ± .004	37.3%	.077 ± .005	.053 ± .001	.147 ± .004	54.6%

Reconstruction. Given $n \in \{1, \ldots, 4\}$ orthographic images, the desired output is a DSL program whose rendered geometry matches the target. Because every model has four views (Section 4.2), each model contributes $\binom{4}{n}$ examples to the n-view dataset.

Material understanding. Given a structure description, the desired output predicts six global properties: Young's modulus E, shear modulus G, bulk modulus K, Poisson ratio ν, anisotropy A, and volume fraction V. Values are rounded to two significant figures. Our benchmark supports two query types: *multiview_and_code* (four images + DSL code) and *single_image* (one image). The relative performance on each type indicates whether additional context helps or hinders a given VLM.

Inverse design. Given a target property profile, the desired output is a DSL program whose simulated properties satisfy the profile. We generate datasets for six query types, where the length-n query requests $n \in \{1, \ldots, 6\}$ property targets per profile. Targets may be exact values, ranges, or upper/lower bounds—e.g., "auxetic ($\nu < 0$)" or "volume fraction $V \approx 0.6$." To construct target profiles from a model, we (1) sample n active properties from the model, (2) choose bounds for each, and (3) render a natural-language prompt using a grammar conditioned on each property's part-of-speech tag (adjective, verb, etc.). This process is detailed in Section G.2. Both the prompt and the underlying numeric targets are stored, so users can rephrase questions or bypass NLP entirely.

5.2 Task-Based Example Format

The query/response pairs are constructed using prompt templates that are specific to each task type (listed in Section I). For a metamaterial and a task type, we gather the data that will be used to construct the query/ground truth response, and the information required to evaluate the predicted response. The intermediate format used to organize this information is detailed in Section G.1. In addition to being model agnostic, this intermediate format allows researchers to reframe prompts without regenerating or deviating from the core content of the inquiry. The intermediate representation also makes MetaBench applicable to traditional non-AI methods. However, since no traditional methods cover the full breadth of MetaBench, we do not include traditional baselines in our evaluations.

6 Results

6.1 Database

MetaDB is, to our knowledge, one of the largest metamaterial databases ever collected, comprising $153,263$ materials. Our dataset features 36,997 expert material designs, including 1,588 variations of 50 hand-authored programs, 1,205 generations, and 34,204 generation parameter variations. These are augmented by 12,029 hybrids and 141,234 mutations. Figure 5 (left) shows that augmentation improves MetaDB's property coverage and gamut, including roughly doubling the range of anisotropies and up to quadrupling directional Poisson ratios.

6.2 Benchmark & Baseline

The 13,282 authored, generated, and hybrid models form the core set from which MetaBench is sampled. We randomly split these models into 500 test, 50 validation, and 12,732 training materials, and generated benchmark tasks for each as described in Section 5 (Figure 5, center). We measured 5 baselines methods on MetaBench: two MetaAssist models, Nova and LLaVA, named after the

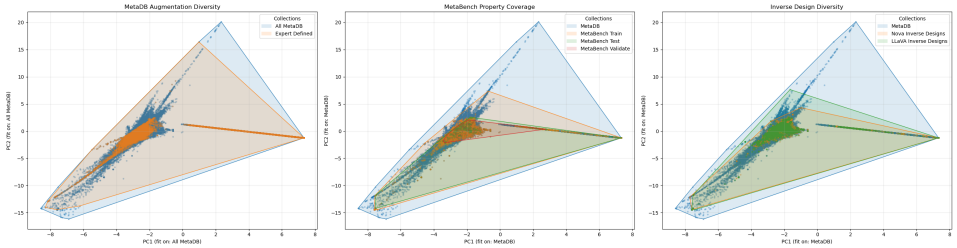


Figure 5: Diversity and Coverage relative to MetaDB. Each image plots the materials properties and their convex hull of MetaDB subsets or MetaAssist generations relative to MetaDB, in MetaDB material property 2D PCA space. Left: MetaDB has a larger gamut (hull) and more dense areas than its expert subset, validating that augmentations extend property range and coverage. Center: Range and coverage of the MetaBench splits. Right: Diversity of MetaAssist generations with the Nova and LLaVA models; both generate materials with a wide gamut of properties.

Base models from which they are tuned on MetaBench (NovaBase and LLaVABase), and Open AI's o3 reasoning model. Training and inference details including full prompt templates are given in Sections H and I. Table 1 summarizes these benchmarks in three categories:

Material Reconstruction 3D structure similarity, measured by intersection over union (IoU) and volumetric chamfer distance of the voxelized unit cells.

Material Understanding Averaged Normalized Error across six properties: anisotropy, Young's modulus (VRH), Bulk Modulus (VRH), Shear Modulus (VRH), Poisson's Ratio, and Volume Fraction, normalized to the typical range of that property across the core material set.

Inverse Design Inverse design is measured by a clipped Averaged Normalized Error. For specific value targets, normalized error is computed as in material understanding. For bounds targets, normalized error is taken relative to the bound (and is zero if the bound is respected).

Both LLaVA- and Nova-based MetaAssist models achieve high material validity rates in generative tasks (reconstruction and inverse design), and low errors. Qualitative understanding of these errors is illustrated by results galleries in Section F. Perhaps suprisingly, the LLaVa model occasionally outperforms the significantly larger Nova model. This is likely to due a post-training step for maintaining general task capabilities (Section J). Ablations (Section F.1) show that general models trained on all tasks outperform task-specific models.

6.3 Interactive Case Studies

We built a metamaterial copilot interface to explore practical scenarios and conducted a series of case studies, using the Nova-variant of MetaAssist as our interactive model due to its large context window and stronger conversational abilities. We experimented with a variety of prompts, and present here a scenario that illustrate the potential of a metamaterial design copilot. Images are compelling input for material design because they cover trying a new material described pictorially in literature, sketching an idea for a design, or taking inspiration from a structure in nature. We prototyped this functionality with a material from the MetaBench test set; even though we presented our request conversationally rather than in structured form, we were still able to obtain and fabricate a perfect reconstruction. A second case study in Section E demonstrates multi-turn editing in an inverse design context.

7 Discussion, Limitations, and Future Work

Metamaterial design is an inherently multimodal, high-impact problem that requires complex reasoning and preference consideration, which makes it a natural test bed for AI development. Conversely, metamaterial researchers have called for better data sets and AI-powered tools. MetaDSL and MetaDB provide a common, traceable descriptor that both communities can adopt. As researchers contribute

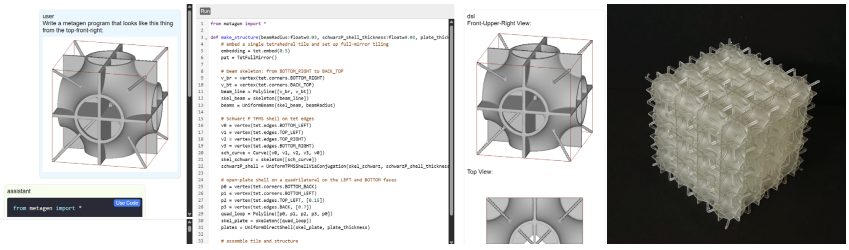


Figure 6: Reconstruction: (Left) Generating a metamaterial program from an input image enables incorporating designs from literature, sketches, and nature. (Right) 3D printed design.

new designs in this format, the database will grow organically, giving machine-learning practitioners richer training data while delivering state-of-the-art design assistants to materials scientists.

Our work provides a comprehensive framework toward these goals, offering myriad opportunities for improvement. We deliberately restricted our MetaAssist implementation to simple supervised fine tuning to provide a bedrock baseline for this new task. This provides common metric for techniques such as RAG to read papers and retrieve patterns, chain-of-thought reasoning to connect design intent to property profiles, and RL training with curriculum learning to generalize to novel inverse design profiles.MetaDSL is designed to be retargetable (Section B), as evidenced by our proof-of-concept extensions. A more flexible geometry kernel would unlock robust non-cubic and aperiodic tilings. Targeting a faster kernel would enable larger and more interactive workflows (e.g. interactive output simulation – we currently often need multiple attempts to get a verifiably correct output), simulation-in-the-loop optimization, and an even-wider data-set scale.

MetaDB also has ample opportunities for growth as a community project, including the implementation of additional generators (Sun et al., 2023a; Liu et al., 2022; Abu-Mualla & Huang, 2024; Makatura et al., 2023), systematic inclusion of singular design templates from metamaterial literature, and diversity-guided synthesis. Our program's explicit semantic structure could support taxonomy construction and intelligent exploration of large design spaces. With broad participation, MetaDB could become the primary resource for tracking metamaterial lineages, structure–property relationships, and mechanistic insights—paralleling the role ImageNet played in computer vision.

At the same time, our framework may encounter misguided application, as our multilayer stack—simulation, code generation, and VLM reasoning—can introduce errors. This deserves particular attention in a domain like metamaterials, which is difficult to intuit about, and an active frontier of science with rapidly changing understanding. The resulting materials may also be deployed in scenarios where inaccurate results may lead to catastrophic failure of engineered products or infrastructure. Thus, results must be validated before deployment, and communications should avoid overstating reliability. Our format already takes small strides toward ensuring the accuracy and traceability of information by including detailed provenance records in each of our models. To further improve transparency, we also release our artifacts and the pipelines used to generate them. Moving forward, it would be prudent to include additional safeguards such as automated validity checks, uncertainty estimates, and safety factors.

8 Conclusion

We introduced **MetaGen**, a unified ecosystem for vision–language metamaterial design that combines (i) *MetaDSL*, a compact yet expressive domain-specific language; (ii) *MetaDB*, an over 150 000-entry database with paired geometry, renderings, and physics; (iii) *MetaBench*, a task-oriented benchmark that probes reconstruction, material understanding, and inverse design; and (iv) *MetaAssist*, the first VLM-driven CAD interface for architected materials. Our baseline experiments illustrate that large vision–language models offer promising performance for multi-modal translation and design generation. Moreover, we provide a holistic vision for accelerated, symbiotic research at the intersection of machine learning and architected materials. With the introduction of MetaGen as both a challenging benchmark for multimodal models and a practical toolkit for materials scientists, our paper lays the foundation to bring this vision to life.

REFERENCES

Mohammad Abu-Mualla and Jida Huang. A dataset generation framework for symmetry-induced mechanical metamaterials. *Journal of Mechanical Design*, 147(4):041705, 12 2024. ISSN 1050-0472. doi: 10.1115/1.4066169.

Ryan P. Adams and Peter Orbanz. Representing and learning functions invariant under crystallographic groups, 2023. URL https://arxiv.org/abs/2306.05261.

Mostafa Akbari, Armin Mirabolghasemi, Mohammad Bolhassani, Abdolhamid Akbarzadeh, and Masoud Akbarzadeh. Strut-based cellular to shellular funicular materials. *Advanced Functional Materials*, 32(14):2109725, 2022. doi: https://doi.org/10.1002/adfm.202109725. URL https://advanced.onlinelibrary.wiley.com/doi/abs/10.1002/adfm.202109725.

Rita Ambu and Anna Eva Morabito. Modeling, assessment, and design of porous cells based on schwartz primitive surface for bone scaffolds. *The Scientific World Journal*, 2019, 2019.

Arash Ataee, Yuncang Li, Darren Fraser, Guangsheng Song, and Cuie Wen. Anisotropic ti-6al-4v gyroid scaffolds manufactured by electron beam melting (ebm) for bone implant applications. *Materials & Design*, 137, 2018.

Reza Attarzadeh, Seyed-Hosein Attarzadeh-Niaki, and Christophe Duwig. Multi-objective optimization of tpms-based heat exchangers for low-temperature waste heat recovery. *Applied Thermal Engineering*, 212, 2022.

Sahab Babaee, Jongmin Shim, James C Weaver, Elizabeth R Chen, Nikita Patel, and Katia Bertoldi. 3d soft metamaterials with negative poisson's ratio. *Advanced Materials*, 25(36), 2013.

J. B. Berger, H. N. G. Wadley, and R. M. McMeeking. Mechanical metamaterials at the theoretical limit of isotropic elastic stiffness. *Nature*, 543(7646):533–537, Mar 2017. ISSN 1476-4687. doi: 10.1038/nature21075. URL https://doi.org/10.1038/nature21075.

Siyuan Bian, Chenghao Xu, Yuliang Xiu, Artur Grigorev, Zhen Liu, Cewu Lu, Michael J Black, and Yao Feng. Chatgarment: Garment estimation, generation and editing via large language models. 2025.

Ramalingaiah Boda, Biranchi Panda, and Shanmugam Kumar. Bioinspired design of isotropic lattices with tunable and controllable anisotropy. *Advanced Engineering Materials*, 27(11):2401881, 2025. doi: https://doi.org/10.1002/adem.202401881. URL https://advanced.onlinelibrary.wiley.com/doi/abs/10.1002/adem.202401881.

Yu-Chin Chan, Faez Ahmed, Liwei Wang, and Wei Chen. METASET: Exploring Shape and Property Spaces for Data-Driven Metamaterials Design. *Journal of Mechanical Design*, 143(3), 2020.

Zeyao Chen, Yi Min Xie, Xian Wu, Zhe Wang, Qing Li, and Shiwei Zhou. On hybrid cellular materials based on triply periodic minimal surfaces with extreme mechanical properties. *Materials & Design*, 183:108109, 2019. ISSN 0264-1275. doi: https://doi.org/10.1016/j.matdes.2019.108109. URL https://www.sciencedirect.com/science/article/pii/S0264127519305477.

Jiin Choi, Seung Won Lee, and Kyung Hoon Hyun. Genpara: Enhancing the 3d design editing process by inferring users' regions of interest with text-conditional shape parameters. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, CHI '25, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400713941. doi: 10.1145/3706598.3713502. URL https://doi.org/10.1145/3706598.3713502.

Zhaohui Fan, Renjing Gao, and Shutian Liu. Thermal conductivity enhancement and thermal saturation elimination designs of battery thermal management system for phase change materials based on triply periodic minimal surface. *Energy*, 259, 2022.

Joseph W. Fisher, Simon W. Miller, Joseph Bartolai, Timothy W. Simpson, and Michael A. Yukish. Catalog of triply periodic minimal surfaces, equation-based lattice structures, and their homogenized property data. *Data in Brief*, 49:109311, 2023. ISSN 2352-3409. doi: https://doi.org/10.1016/j.dib.2023.109311. URL https://www.sciencedirect.com/science/article/pii/S2352340923004298.

Tobias Frenzel, Muamer Kadic, and Martin Wegener. Three-dimensional mechanical metamaterials with a twist. *Science*, 358(6366):1072–1074, 2017. doi: 10.1126/science.aao4640. URL https://www.science.org/doi/abs/10.1126/science.aao4640.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

Ian Huang, Guandao Yang, and Leonidas Guibas. Blenderalchemy: Editing 3d graphics with vision-language models. In *European Conference on Computer Vision*, pp. 297–314. Springer, 2024.

Benjamin Jenett, Christopher Cameron, Filippos Tourlomousis, Alfonso Parra Rubio, Megan Ochalek, and Neil Gershenfeld. Discretely assembled mechanical metamaterials. *Science Advances*, 6(47), 2020.

R Kenny Jones, Paul Guerrero, Niloy J Mitra, and Daniel Ritchie. Shapelib: designing a library of procedural 3d shape abstractions with large language models. *arXiv preprint arXiv:2502.08884*, 2025.

Vikram Kumaran, Jonathan Rowe, Bradford Mott, and James Lester. Scenecraft: automating interactive narrative scene generation in digital games with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 19, pp. 86–96, 2023.

Doksoo Lee, Wei (Wayne) Chen, Liwei Wang, Yu-Chin Chan, and Wei Chen. Data-driven design for metamaterials and multiscale systems: A review. *Advanced Materials*, 36(8):2305254, 2024. doi: https://doi.org/10.1002/adma.202305254. URL https://advanced.onlinelibrary.wiley.com/doi/abs/10.1002/adma.202305254.

Beichen Li, Rundi Wu, Armando Solar-Lezama, Liang Shi, Changxi Zheng, Bernd Bickel, and Wojciech Matusik. VLMaterial: Procedural material generation with large vision-language models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=wHebuIb6IH.

Feng Li, Renrui Zhang, Hao Zhang, Yuanhan Zhang, Bo Li, Wei Li, Zejun Ma, and Chunyuan Li. Llava-next-interleave: Tackling multi-image, video, and 3d in large multimodal models. *arXiv preprint arXiv:2407.07895*, 2024.

Haotian Liu, Chunyuan Li, Yuheng Li, Bo Li, Yuanhan Zhang, Sheng Shen, and Yong Jae Lee. Llava-next: Improved reasoning, ocr, and world knowledge, January 2024. URL https://llava-vl.github.io/blog/2024-01-30-llava-next/.

Peiqing Liu, Bingteng Sun, Jikai Liu, and Lin Lu. Parametric shell lattice with tailored mechanical properties. *Additive Manufacturing*, 60:103258, 2022. ISSN 2214-8604. doi: https://doi.org/10.1016/j.addma.2022.103258.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Liane Makatura, Bohan Wang, Yi-Lu Chen, Bolei Deng, Chris Wojtan, Bernd Bickel, and Wojciech Matusik. Procedural metamaterials: A unified procedural graph for metamaterial design. *ACM Trans. Graph.*, 42(5), July 2023. ISSN 0730-0301. doi: 10.1145/3605389.

Liane Makatura, Michael Foshey, Bohan Wang, Felix Hähnlein, Pingchuan Ma, Bolei Deng, Megan Tjandrasuwita, Andrew Spielberg, Crystal Owens, Peter Yichen Chen, Allan Zhao, Amy Zhu, Wil Norton, Edward Gu, Joshua Jacob, Yifei Li, Adriana Schulz, and Wojciech Matusik. How Can Large Language Models Help Humans in Design and Manufacturing? Part 1: Elements of the LLM-Enabled Computational Design and Manufacturing Pipeline. *Harvard Data Science Review*, (Special Issue 5), dec 23 2024a. https://hdsr.mitpress.mit.edu/pub/15nqmdzl.

Liane Makatura, Michael Foshey, Bohan Wang, Felix Hähnlein, Pingchuan Ma, Bolei Deng, Megan Tjandrasuwita, Andrew Spielberg, Crystal Owens, Peter Yichen Chen, Allan Zhao, Amy Zhu, Wil Norton, Edward Gu, Joshua Jacob, Yifei Li, Adriana Schulz, and Wojciech Matusik. How Can Large Language Models Help Humans in Design And Manufacturing? Part 2: Synthesizing an End-to-End LLM-Enabled Design and Manufacturing Workflow. *Harvard Data Science Review*, (Special Issue 5), dec 23 2024b. https://hdsr.mitpress.mit.edu/pub/hiii8fyn.

Timon Meier, Vasileios Korakis, Brian W. Blankenship, Haotian Lu, Eudokia Kyriakou, Savvas Papamakarios, Zacharias Vangelatos, M. Erden Yildizdag, Gordon Zyla, Xiaoxing Xia, Xiaoyu Zheng, Yoonsoo Rho, Maria Farsari, and Costas P. Grigoropoulos. Scalable phononic metamaterials: Tunable bandgap design and multi-scale experimental validation. *Materials & Design*, 252: 113778, 2025. ISSN 0264-1275. doi: https://doi.org/10.1016/j.matdes.2025.113778. URL https://www.sciencedirect.com/science/article/pii/S0264127525001984.

Mehran Mirramezani, Anne S. Meeussen, Katia Bertoldi, Peter Orbanz, and Ryan P Adams. Designing mechanical meta-materials by learning equivariant flows. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=VMurwgAFWP.

Muhammad and C. W. Lim. Phononic metastructures with ultrawide low frequency three-dimensional bandgaps as broadband low frequency filter. *Scientific Reports*, 11(1), 2021.

Kiyohiro Nakayama, Jan Ackermann, Timur Levent Kesdogan, Yang Zheng, Maria Korosteleva, Olga Sorkine-Hornung, Leonidas Guibas, Guandao Yang, and Gordon Wetzstein. Aipparel: A large multimodal generative model for digital garments. *Computer Vision and Pattern Recognition (CVPR)*, 2025.

Julian Panetta, Qingnan Zhou, Luigi Malomo, Nico Pietroni, Paolo Cignoni, and Denis Zorin. Elastic textures for additive fabrication. *ACM Trans. Graph.*, 34(4), July 2015. ISSN 0730-0301. doi: 10.1145/2766937.

Zhao Qin, Gang Seob Jung, Min Jeong Kang, and Markus J. Buehler. The mechanics and design of a lightweight three-dimensional graphene assembly. *Science Advances*, 3(1), 2017.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, Jan 2024. ISSN 1476-4687. doi: 10.1038/s41586-023-06924-6. URL https://doi.org/10.1038/s41586-023-06924-6.

Bingteng Sun, Xin Yan, Peiqing Liu, Yang Xia, and Lin Lu. Parametric plate lattices: Modeling and optimization of plate lattices with superior mechanical properties. *Additive Manufacturing*, 72: 103626, 2023a. ISSN 2214-8604. doi: https://doi.org/10.1016/j.addma.2023.103626.

Chunyi Sun, Junlin Han, Weijian Deng, Xinlong Wang, Zishan Qin, and Stephen Gould. 3d-gpt: Procedural 3d modeling with large language models. *arXiv preprint arXiv:2310.12945*, 2023b.

James Utama Surjadi, Bastien F G Aymon, Molly Carton, and Carlos M Portela. Double-network-inspired mechanical metamaterials. *Nat Mater*, April 2025.

Zhengyi Wang, Jonathan Lorraine, Yikai Wang, Hang Su, Jun Zhu, Sanja Fidler, and Xiaohui Zeng. Llama-mesh: Unifying 3d mesh generation with language models. *arXiv preprint arXiv:2411.09595*, 2024.

Benjamin C. White, Anthony Garland, Ryan Alberdi, and Brad L. Boyce. Interpenetrating lattices with enhanced mechanical functionality. *Additive Manufacturing*, 38:101741, 2021. ISSN 2214-8604. doi: https://doi.org/10.1016/j.addma.2020.101741. URL https://www.sciencedirect.com/science/article/pii/S2214860420311131.

Yonglai Xu, Hao Pan, Ruonan Wang, Qiang Du, and Lin Lu. New families of triply periodic minimal surface-like shell lattices. *Additive Manufacturing*, 77:103779, 2023. ISSN 2214-8604. doi: https://doi.org/10.1016/j.addma.2023.103779.

Tianyang Xue, Haochen Li, Longdu Liu, Paul Henderson, Pengbin Tang, Lin Lu, Jikai Liu, Haisen Zhao, Hao Peng, and Bernd Bickel. Mind: Microstructure inverse design with generative hybrid neural representation, 2025. URL https://arxiv.org/abs/2502.02607.

Yutaro Yamada, Khyathi Chandu, Yuchen Lin, Jack Hessel, Ilker Yildirim, and Yejin Choi. L3go: Language agents with chain-of-3d-thoughts for generating unconventional objects. *arXiv preprint arXiv:2402.09052*, 2024.

Yanyan Yang, Lili Wang, Xiaoya Zhai, Kai Chen, Wenming Wu, Yunkai Zhao, Ligang Liu, and Xiao-Ming Fu. Guided diffusion for fast inverse design of density-based mechanical metamaterials, 2024a. URL https://arxiv.org/abs/2401.13570.

Yue Yang, Fan-Yun Sun, Luca Weihs, Eli VanderBilt, Alvaro Herrasti, Winson Han, Jiajun Wu, Nick Haber, Ranjay Krishna, Lingjie Liu, et al. Holodeck: Language guided generation of 3d embodied ai environments. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 16227–16237, 2024b.

Zhenze Yang and Markus J. Buehler. High-throughput generation of 3d graphene metamaterials and property quantification using machine learning. *Small Methods*, 6(9):2200537, 2022. doi: https://doi.org/10.1002/smtd.202200537.

Haocheng Yuan, Jing Xu, Hao Pan, Adrien Bousseau, Niloy J. Mitra, and Changjian Li. Cadtalk: An algorithm and benchmark for semantic commenting of cad programs. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3753–3762, 2024. doi: 10.1109/CVPR52733.2024.00360.

Feng Zhou, Ruiyang Liu, Chen Liu, Gaofeng He, Yong-Lu Li, Xiaogang Jin, and Huamin Wang. Design2garmentcode: Turning design concepts to tangible garments through program synthesis. 2025.

Ata Çelen, Guo Han, Konrad Schindler, Luc Van Gool, Iro Armeni, Anton Obukhov, and Xi Wang. I-design: Personalized llm interior designer, 2024.

A Appendix

B Ecosystem Design

The four components of the MetaGen ecosystem work together to achieve our design goals. We outline these goals and the design and organization decisions that achieve them here:

- MetaDB
  - Design Goals: Collect existing knowledge in a reconfigurable, reusable, and task independent manner
  - Organization
    - Primary Elements: Material Definitions; Provenance
    - Derived Elements: Geometry; Computed Properties
- MetaBench
  - Design Goals:
  - Organization:
    - Primary Elements: Structured Task Definitions; Target Data; References, Evaluation Procedures
    - Derived Elements: Query Strings; Example Responses
- MetaDSL
  - Design Goals: Eventual Comprehensiveness via Extensibility; Supports Hybrid Structures Easily; Ease of Use
  - Design Decisions: Extensible Embedded Python DSL for extensibility and Ease-of-Us; Separation of Front-End Language from Geometry Kernel

- MetaAssist
  - Design Goals: Usable for general engineers; single interface across design silos; possibility of integrating unstructured data (literature, sketches, etc.)
  - Elements: Interactive Interface; Trained Baseline Models

Each component supports the others, as illustrated in Figure 7

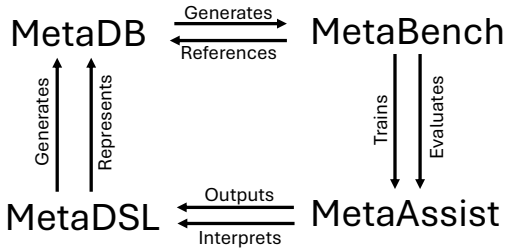


Figure 7: Relationships between MetaGen ecosystem components.

B.1 Ecosystem Development and Insights

The elements of this ecosystem were developed in concert with one another, going through 3 major iterations before arriving at their current state. MetaDSL was at the heart of each iteration, as the representation has a direct impact on the efficacy of the other three components:

- MetaDB needs a representation that captures diverse structures, but also offers robust pathways for scalable (and, in this case, VLM-driven) structure generation, hybridization, mutation, sampling, etc.
- MetaBench can only be used for training and evaluation if it is built atop a large, diverse database.
- MetaAssist relies on a strong training corpus from MetaBench. MetaAssist also hinges on the intelligibility of the representation, and the model's ability to interpret, generate, and modify programs according to user input.

We defer the language-specific development details to Section C.4.

Outside the scope of the DSL, we also found that dataset management and curation posed a major hurdle. We improved diversity by continuously mining metamaterial literature for additional seed program designs. We expressed these seed programs as-parametrically-as-possible to allow for expert-driven sampling. As we scaled the dataset, we also realized that it would be critical to keep track of the programs' sources and relationship to one another. This information is especially useful for navigation, contextualization and diversity management, particularly as the database grows in response to community effort. To manage this, we introduced a formalized provenance system for MetaDB.

C MetaDSL

C.1 Additional Implementation Details

We implemented the core functionality of MetaDSL (version 1.1.0) with two goals in mind. First, we wanted full support for the metamaterials that were expressible in our geometry kernel, ProcMeta. Second, we wanted our infrastructure to easily permit extensions in the future without invalidating existing programs. We detail the current state of each feature category in our language: convex polytopes, skeletons, lifting procedures, tiles, and patterns. For a full API description of the accessible functions, please refer to Section I.2. Figure 8 shows an overview of the compiler architecture.

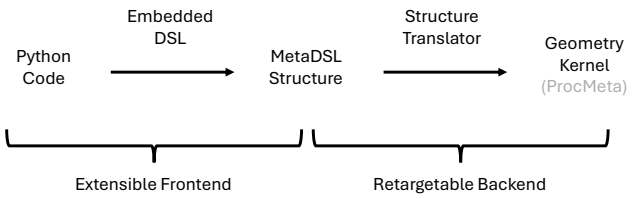


Figure 8: Overview of MetaDSL's implementation. MetaDSL programs are written in an embedded Python DSL frontend to allow for ease of use and extensibility. These structures are compiled into a structured intermediate representation, and a backend Translator converts these structures into geometry kernel instructions. In our implementation we used the geometry kernel from ProcMeta Makatura et al. (2023). By separating the front-end representation from the backend geometry kernel, MetaDSL is flexible to both be extended in its frontend representation, and retargettable to different geometry backends for new applications, while keeping a compatible material representation.

Convex Polytopes (CP) Currently, all of our programs make use of three pre-defined CPs (as inspired by ProcMeta): `cuboid`, `triPrism` and `tet`. The infrastructure to define custom convex polytopes exists, and most operators up to and including Tiles should generalize to such CPs. However, the patterning operations would need to be generalized before being able to operate on arbitrary CPs.

Skeletons Then, a *skeleton* is constructed via a set of vertices and edges that are positioned relative to a common CP. Each vertex is positioned on a particular CP entity (corner, edge, face, interior). Each CP entity is accessed via a semantically meaningful alias, permitting calls such as e.g. `vertex(cuboid.edges.BACK_LEFT)`. The `vertex` call also optionally takes a list $\vec{t}$ of interpolation values used to position the vertex within the entity. If $\vec{t}$ is omitted, the returned point will be at the entity's midpoint (edge) or centroid (face/interior). Presently, corners ignore weights (since they cannot be moved); edges use linear interpolation; and faces use barycentric coordinates if they contain 3 vertices or bilinear interpolation for quads. If a CP with different polygonal faces (e.g. pentagons) were implemented, an appropriate lower-dimensional vertex positioning specification would need to be devised. Internally, the vertices are stored using weights over a full list of the CP corners, so additional specification interfaces can easily be defined.

An ordered list of vertices can then be strung together into simple (non-branching, self-intersection-free) open or closed paths via the `Polyline` or `Curve` commands. Each edge contained in a path infers and maintains information about its incidence on the CP – including whether it is contained within a face, through the CP volume, coincident with a CP edge, etc. This is very useful when determining lifting function compatibility, as some procedures can only be applied when e.g. every path edge is contained within a CP face.

Then, a `skeleton` is used to combine a set of vertices or polylines/curves into a larger, more complex element, over which additional organizational information is computed. Skeletons infer the connected components formed by the inputs, then categorize them based on their topology. Thus, a skeleton may be labeled as a simple closed loop, even if the input is a set of open paths. Again, these insights are critical for determining the skeleton's compatibility with downstream operations, such as lifting procedures. We also included infrastructure for the skeletons to infer and track their total incidence on each entity of the reference CP, including the dimensionality (e.g. point or line) of an intersection – however, this feature is not fully implemented in the current MetaDSL version.

Lifting Procedures *Lifting procedures* are used to transform the skeleton into a volumetric object. Simple procedures like `Spheres` instantiate a sphere of the given radius centered at each vertex in the skeleton. Similarly, `UniformBeams` instantiates a beam of the given thickness centered along each path of the input skeleton. The shell operators (`UniformDirectShell`, `UniformTPMSShellViaMixedMinimal`, and `UniformTPMSShellViaConjugation`) solve for a surface that spans the provided boundary curve before expanding the surface to the desired thickness. Our shell and beam procedures mimic those defined by ProcMeta, as they cover

a wide range of metamaterial classes and were already (by construction) natively supported by our geometry kernel. Our `Curve` and `Polyline` commands correspond to their smooth/non-smooth edge chains, respectively. Unlike the original, we chose to explicitly separate several operators that were previously lumped together, which clarified and minimized the number of exposed parameters for each call.

Tiles To create an embedded, patternable tile, we provide a list of one or more lifted skeletons as input to the `Tile` operator. The tile operator also takes as input the embedding information, which will be used to embed the CP and, in turn, each vertex of the contained skeleton(s). To obtain the embedding information, each CP implements at least one `embed` function, which takes high level parameters such as the min/max position of the CP's AABB.

Because of constraints imposed by $ProcMeta$ – that these must form a partition of the unit cell – our code currently treats these CPs with some additional assumptions. Specifically, though the cuboid need not be a cube, it must have right angles everywhere, and edge lengths must be $1/2^k$ for some positive integer k; in practice, $k \in [1, ..4]$. The triPrism is assumed to be an isoceles triangle with a right angle. The tet similarly has a base that is an isoceles triangle with a right angle, and a fourth vertex that is located directly above one of the 45 degree angles. These assumptions would ideally be relaxed in a future version of MetaDSL.

Patterns Patterns are currently the most restricted feature of MetaDSL, as we restrict our dataset to programs that can be compiled down to the language and solver set described by ProcMeta. Thus, rather than extending our structures to a more arbitrary tiling in $\mathbb{R}^3$, all of our structures have a translational unit residing in a unit cube. The pattern operators were written in a way that allows for additional, extended tiling procedures. We prioritized mirrors, because they are sufficient to express a wide range of common metamaterial designs, and they are often used in generative metamaterial design schemes, as the connectivity requirements are simpler than most other operations. We also have limited support for other operations such as `Rotate180` and `Translate`, which can be used inside the `Custom` pattern specifier. Currently, these limited operations are only defined for specific transformations on cuboids. We look forward to an expanded MetaDSL that includes full support for these patterning operations, at least over the pre-built CPs that currently exist. In the long term, we envision a patterning system that extends well beyond this, to support large, potentially aperiodic or asymmetric tilings composed of one or more tiles with arbitrary CPs. This is a very difficult problem, and will itself present an interesting set of research directions, including how to intuitively specify these patterns and how to characterize their compatibility/validity.

C.2 Example Programs

Example program-structure pairs are listed in Figure 9 and Figure 10. Many additional models can be found in the accompanying data.

C.3 MetaDSL vs. ProcMeta

As suggested by Section B.1 and the architecture diagram in Figure 8, MetaDSL is distinct from and strictly more general than ProcMeta, with a design philosophy all its own. Our approach was motivated by our early experiments with ProcMeta, which revealed a critical shortcoming: important information was represented implicitly in the ProcMeta GUI interface, and was entirely absent from the ProcMeta graph representation. To make this information accessible to LLMs (and more easily accessible to humans), we implemented a programmatic interface, MetaDSL, that compiles to the same geometry kernel as ProcMeta, but provides several practical advantages (see Table 2).

Most importantly, MetaDSL introduces explicit, referenceable bounding volumes (BVs), which are critical for verifying and enforcing the preconditions of geometry operations. In the ProcMeta GUI, BVs exist only as non-referenceable visual aids; users must manually align coordinates, and no automated compatibility checks are possible. ProcMeta graphs omit BVs entirely. MetaDSL represents BVs through a CP abstraction, which enforces constraints by construction, enables type checking, and cleanly separates tile content from patterning, improving modularity and reconfigurability. These features align the representation more closely with the valid shape space, aiding both human designers and LLMs in producing valid, diverse structures. MetaDSL programs also make heavy use

```
from metagen import *

def make_structure(shell_thickness=0.03) -> Structure:
    v0 = vertex(tet.edges.BOTTOM_LEFT)
    v1 = vertex(tet.edges.TOP_LEFT)
    v2 = vertex(tet.edges.TOP_RIGHT)
    v3 = vertex(tet.edges.BOTTOM_RIGHT)

    c0 = Curve([v0, v1, v2, v3, v0])

    skel = skeleton([c0])
    shell = UniformTPMSShellViaConjugation(skel, shell_thickness)

    embedding = tet.embed(0.5)
    tile = Tile([shell], embedding)
    pat = TetFullMirror()
    obj = Structure(tile, pat)

    return obj
```

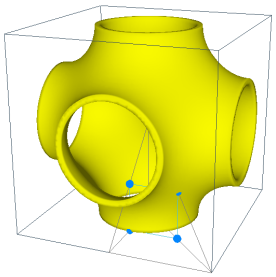


Figure 9: Example program and corresponding geometry for the Schwarz P structure.

```
from metagen import *

def make_structure(beamRadius_narrow=0.03, beamRadius_wide=0.1) -> Structure:
    embed = cuboid.embed(0.5, 0.5, 0.5,
                         cornerAtAABBMin=cuboid.corners.FRONT_BOTTOM_LEFT)

    v0 = vertex(cuboid.corners.FRONT_BOTTOM_LEFT)
    v1 = vertex(cuboid.corners.BACK_TOP_RIGHT)
    p0 = Polyline([v0, v1])

    skel = skeleton([p0])
    liftedSkel = SpatiallyVaryingBeams(skel, [[0, beamRadius_narrow],
                                              [0.5, beamRadius_wide],
                                              [1, beamRadius_narrow]])

    tile = Tile([liftedSkel], embed)
    pat = Custom(Rotate180([cuboid.edges.BACK_RIGHT,
                            cuboid.edges.BACK_LEFT], True,
                           Rotate180([cuboid.edges.TOP_RIGHT], True)))
    obj = Structure(tile, pat)

    return obj
```

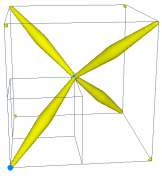


Figure 10: Example program and corresponding geometry for the pentamode structure.

	MetaDSL	ProcMeta
Compactness	**Shorter**, less boilerplate. Easier to read, less likely to exceed token limits	Longer, more boilerplate. Exceeds context of small, lightweight models.
Modules	**Highly reusable**. Patterns defined in composable chunks (eg TetMirror), independent of tile contents. Skeletons defined independent of embedding, easily scale to different Tiles.	**No support.** Limited reuse. Patterns can't exist independently; no pre-built Patterns. Absolute Skeletons, cannot easily be rescaled.
Relative vs. Absolute Positioning	Positions and transforms use **local coordinates** (i.e. [0,1]) wrt named entities (`cuboid.edges.TOP_LEFT`) in abstract polytopes. Robust for generation, clear design space bounds, more intuitive.	Positions and transforms use **absolute coordinates**. Easily misaligned, difficult to visualize without plotting. Unsuitable for VLMs, which struggle with computation/spatial tasks.
BV representation	**Explicit BV with named, referenceable entities.** Facilitates verifiable parametric design, e.g., vertex constrained to given BV edge. Allows type/error checking.	**Implicit or Absent BV**: drawn as a visual aid in the GUI, but not represented/preserved in the graph. Never referenceable.
Type/Error checking	**Type/incidence tracking to ensure compatibility** – e.g. conjugate TPMS require a closed loop where every edge lies in a BV face, and every BV face contains at least 1 loop edge. This is known from our representation and verified by downstream operations. Helps determine valid substitutions for mutations, even when large changes are proposed, leading to greater diversity. Critical for complex patterning, to determine compatibility of proposed-adjacent faces.	**None.** The burden of verification (for e.g. vertices on BV edges or edges in BV faces) is left to the user – infeasible for agentic design. Bad inputs crash ProcMeta with no explanation or suggested improvements.
Simplified Operations	**Abstractions simplify element creation**; e.g., Sphere() takes a center point and a radius, as one would expect. Easier for humans and LLMs.	Strict compliance with the given graph interface makes **some operations cumbersome**; e.g. for a sphere, thicken a 0-length edge chain over 2 co-located vertices
Semantic information	**Complete support.** Comments and meaningful variable names improve readability and admit metadata (provenance, parameter bounds)	No support.
Parameters	**Complete support.** Allows parametrized models and family generators.	**None.** Explicit positions etc. only. Variations defined as separate graphs. Difficult/impossible to infer constraints or design space from the graph description.
Loops, Functions	**Supports complex logic** that would be tedious to implement otherwise. Functions are especially useful for hybridization, as programs can be directly reused and/or rescaled.	**No support.** Each instance must be created/connected individually. Even hybridization is difficult, because subgraphs cannot be inserted directly – the identifier/references of each node must be updated.

Table 2: Detailed differences between the interfaces for MetaDSL and ProcMeta.

of programmatic features absent from ProcMeta graphs. Semantic variable names, comments (avg. 4/program), and parametric variables improve human interpretability and support natural-language reasoning for LLMs. Loops and helper functions are also common, appearing in 1,744 and 2,103 of the 13,284 core programs respectively. These features allow compact, self-consistent definitions that would be unwieldy if unrolled or inlined into a ProcMeta graph.

We tested LLM-based augmentation using ProcMeta JSON instead of MetaDSL. MetaDSL yielded: (1) higher code validity (75% vs. 54%), (2) more structurally focused reasoning rather than boilerplate handling, and (3) lower token usage (580 vs. 1,049 tokens on average for o4). Beyond these immediate benefits for LLM usage and dataset generation, our DSL interface also makes MetaDSL a more flexible platform from which to build further extensions, which facilitates its intended purpose as the seed of a wider community project.

Extensibility The MetaDSL interface naturally generalizes to shape spaces that would be difficult to represent in ProcMeta's graph approach. For example, implicit functions are common in metamaterial design, but they would be cumbersome to represent in ProcMeta's graph. However, MetaDSL could naturally include them: rather than an explicit Skeleton, we could use the implicit function to define a SkeletonGenerator; this could then be fed to an Implicit lifting function, which would solidify a given isovalue range. Non-trivial patterning would also be possible through MetaDSL's Custom pattern interface. For example, given a set of mutually compatible unit cells (like the left/right faces of Figure 2a,b,c,f), simple translations could combine them into an elongated, interleaved tile (e.g. ABCCBA). With enhanced compatibility determination, we could also create Pattern procedures for scholastic or aperiodic tilings. This will allow MetaDSL to expand alongside developments in metamaterial design.

We implemented a proof-of-concept for both of these extensions. An example is available in Figure 11.

```
from metagen import *
from sdf import *
from tpms_helpers import *
from common_tpms import gyroid

def make_structure(isoval_min: float =-0.2, isoval_max: float =0.2, l:
    float =1.0) -> Structure:

    cv = CartesianVolume(   cuboid.corners.FRONT_BOTTOM_RIGHT,
                            cuboid.corners.FRONT_BOTTOM_LEFT,
                                1,
                            cuboid.corners.FRONT_TOP_RIGHT,    1,
                            cuboid.corners.BACK_BOTTOM_RIGHT,
                                1)

    shell_sdf = sheet_isosurface_pair(gyroid, isoval_min, isoval_max
        )
    shell = cv.liftedSkelsFromSDF(shell_sdf)
    print(f"Num ccs: {shell[0].skel.num_connected_components()}")
    print(f"Some cc on all faces: {shell[0].skel.
        is_some_cc_on_all_faces()}")

    # embedding and tiling
    side_len = 1.0
    embedding = cuboid.embed(0.5*side_len, side_len, 2*side_len)
    tile = Tile(shell, embedding)
    pat = Custom(Translate(cube.faces.FRONT, cube.faces.BACK, True,
                    Translate(cube.faces.BOTTOM, cube.faces.TOP,
                        True,
                        Translate(cube.faces.LEFT, cube.faces.
                            RIGHT, True))))
    return Structure(tile, pat)
```

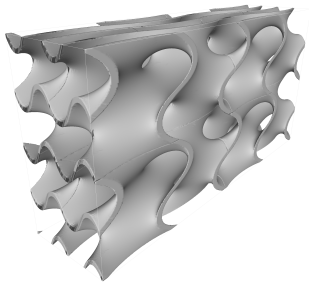


Figure 11: Example program and corresponding geometry for the implicit gyroid, with a stretched unit cell generated by embedding the Tile with a non-unit aspect ratio.

C.4 Language Development Process and Insights

As mentioned in Section B.1, our geometry representation went through 3 major stages.

In the first iteration, we represented metamaterials using ProcMeta graphs directly. This had several issues: it was not compact enough for the context windows of small, lightweight models; intuitiveness and editability suffered dramatically without the aid of a GUI editing tool; the graphs' use of absolute coordinates proved challenging for LLMs (which struggle with spatial reasoning); and the program manipulations (e.g. hybridization, mutation) were unwieldy and fragile, with low validity rates that prohibited effective dataset scaling and diversification. This limited the breadth of MetaDB and MetaBench, while curtailing the efficacy of MetaAssist.

To address this, we designed a higher-level language that became MetaDSL-v0. This approach had a compact, modular, bilevel design that was embedded within Python and thus permitted semantically meaningful content; as such, it solved the context length and human editability issues of ProcMeta. It allowed for relative positioning, which mitigated the issues with coordinates while improving components' reusability. It also allowed for dataset augmentation through programmatic mutation, and improved the efficacy of VLM-based hybridization and mutation – we attributed this jump to our Python embedding, as VLMs show great facility with Python. Still, MetaDSL-v0 remained fragile: generated programs frequently failed, and database augmentations showed limited diversity.

Analysis of MetaDSL-v0's failure modes offered several insights; we arrived at the current MetaDSL by addressing each in turn. First, we noticed that VLMs often used hallucinated synonyms, such as `TOP_LEFT` vs `LEFT_TOP`; we added overloads for all reasonable variations of our functions and attributes. We also found that it was critical to abrogate as much spatial reasoning from the VLM as possible: a full 1/3 of failures were due to the VLM's improper positioning of vertices that form the concrete polytope tiles. We circumvented this through abstracted tile embedding functions, which generate valid embeddings from simple, meaningful parameterizations. In our final large-scale change, we swapped the relative order of lifting functions and tile embeddings (previously Embed then Lift; now, Lift then Embed). This change improved the modularity and compositionality while reducing verbosity – for example, this change allows multiple skeletons to reside in a shared Tile embedding, such that they can be patterned as a single unit. This change also paved the way for patterning of more diverse geometry-generation methods in future extensions. As a result, MetaDSL showed dramatic improvements in generation/mutation rates, and – in turn – significantly more diverse LLM-driven hybridizations.

D MetaDB

D.1 Database Layout

MetaDB is structured into 4 primary directories:

- literature: Literature references that are the sources for hand-authored models.
- models: MetaDSL programs and their outputs.
- generators: Programs that create and augment models
- benchmark: The MetaBench benchmark

Data items in MetaDB can reference other items by path. These paths are either absolute (start with a forward slash "/") or relative (no leading slash). Absolute paths are assumed to start at the root of the database structure. For example, a model may reference the paper that defined it in its sources as `/literature/...`.

D.2 Provenance Information

Each Model in MetaDB starts with a triple-single-quote (`'''`) delimited yaml string called the header-block. This contains useful metadata about the program, including provenance information about how it was created, and what sources it draws on. Provenance information is recorded in two places in the header block.

The primary location is in the "sources" key. This is a dictionary where the keys are MetaDB paths to literature, models, or generators that are the source of this model. The secondary location is in `file_info→generator_info`. For models that are autogenerated via enumeration or augmentation this section contains a MetaDB path to the script that generated the file, the arguments that were passed into that script, and specific `structure_details` that specified this particular model.

D.3 Hybridization Implementation

We hybridized hand-authored models using calls to OpenAI's o4-mini model using a reasoning effort of "medium". For every pair and triplet of authored models, we used the following prompt template:

```
You have access to a DSL whose specification is as follows:
{ api_description }

I want you to help discover unique new programs. Do this by genetic crossover based on these
    parent Metagen DSL programs:

1)
```python
{program 1 code}
```

2)
```python
{program 2 code}
```

Combine relevant structural / logical features from each sample into one coherent DSL program.
Be sure to:
- Respect the DSL syntax strictly.
- Maintain correctness in the final structure definition.
- Keep the final program well-formed and ready to be run as a standard Metagen DSL generator.
- Provide minimal descriptive comments.

Return only the resulting code in a single code block.
```

where `api_description` is the MetaDSL API specification given in Section I, and the program code is listed excluding the header block.

D.4 Mutation Implementation

Our mutation script loads a DSL model from file and constructs the corresponding Structure object in memory. Then, it is able to modify the structure along 4 different axes. Two of the axes allow discrete adjustments: (1) switching any `Polyline` to a `Curve` or vice versa; and (2) selecting a different lifting procedure from the set of options compatible with the skeleton (as inferred by our type system). The remaining modification axes permit continuous variations: (3) repositioning a vertex within its CP element; and (4) selecting a different thickness specification for any lifting procedures. To generate a given variant, each modification axis was permitted with a pre-specified probability; we used $\Pr = 0.7$ for both discrete changes, $\Pr = 0.9$ for vertex perturbation, and $\Pr = 0.98$ for thickness perturbation. Once a given perturbation category was permitted, we looped over each opportunity for said modification within our structure specification, and evaluated a random number against the same respective probability to decide whether this specific instance should be modified or not. For example, with $\Pr = 0.7$ we allow `Polyline`/`Curve` swaps in the variant; then, each time a candidate `Polyline`/`Curve` is identified, we enact the swap with $\Pr = 0.7$. Once an instance has been approved, the specific replacement value was chosen at random from the appropriate set of options (if more than one available). The updated structure is then written to file using the `dslTranslator`, which writes a DSL model from a Structure object. Additional mutation procedures could be implemented to further increase the vawriety of resulting structures.

Provenance Information is stored in the `sources` section of each program's header block. This is a dictionary where the keys are database paths.

D.5 Material Properties

Our simulation provides the 6×6 elastic tensor C in Voigt notation, along with the compliance matrix, $S = C^{-1}$. From this, we extract 18 common material properties:

- E: Young's Modulus, Voigt-Reuss-Hill (VRH) average, relative to E_{base}.
- E_1, E_2, E_3: Directional Young's Moduli, relative to E_{base}
- G: Shear Modulus (VRH average), relative to E_{base}
- G_{23}, G_{13}, G_{12}: Directional Shear Moduli, relative to E_{base}
- ν: Poisson ratio (VRH average)
- $\nu_{12}, \nu_{13}, \nu_{23}, \nu_{21}, \nu_{31}, \nu_{32}$: Directional Poisson ratios
- K: Bulk modulus (VRH average), relative to E_{base}
- A: Anisotropy (universal anisotropy index)
- V: Volume Fraction.

D.6 Ensuring MetaDB Quality

MetaDB is founded on a strong basis of expert programs, including 50 hand-authored examples sourced from diverse, singularly-developed designs in metamaterial literature. This large, diverse collection of seeds is unique to MetaDB, as most large datasets are derived exclusively from a small set of procedural generators. For example, Xue et al. (2025) creates a database of 180k samples, 78% of which stem from variations of the topologies in Elastic Textures (Panetta et al., 2015). The remaining 22% stem from similar generators for planar- and curved-shell structures (Liu et al., 2022; Sun et al., 2023a). Because of the reliance on such generators, Xue et al. (2025) does not offer any representation of e.g. CSG-style structures like the Bucklicrystal of Babaee et al. (2013). However, the bucklicrystal is part of our database, as shown in Figure 4(i), center). MetaDB also already includes Elastic Textures, and similar generators could be implemented for the remaining sources mentioned above.

To ensure that MetaDB only contains high-quality material definitions – even when automatically generating a large portion of our entries – material models are only added after they have passed a series of basic checks. Presently, this includes 3 criteria:

- **MetaDSL compilation:** the model must contain valid python code that successfully evaluates to a MetaDSL Structure object. This includes all runtime type checking done by MetaDSL.
- **Valid Geometry Generation:** after the MetaDSL Structure object is transpiled into the target geometry kernel (in our case, ProcMeta), the kernel is run. We check the resulting geometry for validity, as measured by a non-null result that is tilable in 3D. To determine tilability, we tile the base cell in a $3 \times 3 \times 3$ lattice, then check that the boundaries are periodic and that at least one connected component of this larger base cell reaches all boundaries.
- **Physically Consistent Simulation Results:** the simulator must return reasonable results that obey physical constraints. For example, since our simulation is normalized by the base material's Young's modulus E_{base}, it must be the case that our simulation returns $E \leq 1$.

D.7 MetaDB Statistics

MetaDB covers a wide range of material properties, illustrated here in as histograms (Figure 12) and as parallel coordinates (Figure 13). MetaDB has dense coverage over most of its range of elastic moduli, mid-range coverage of Poisson ratios, and dense coverage of low-anisotropy materials.

E Additional Case Study

Here we present a second case study in iterative inverse design. In Figure 14, we specify a set of target property bounds, and the model is able to generate a metamaterial that satisfies them (we

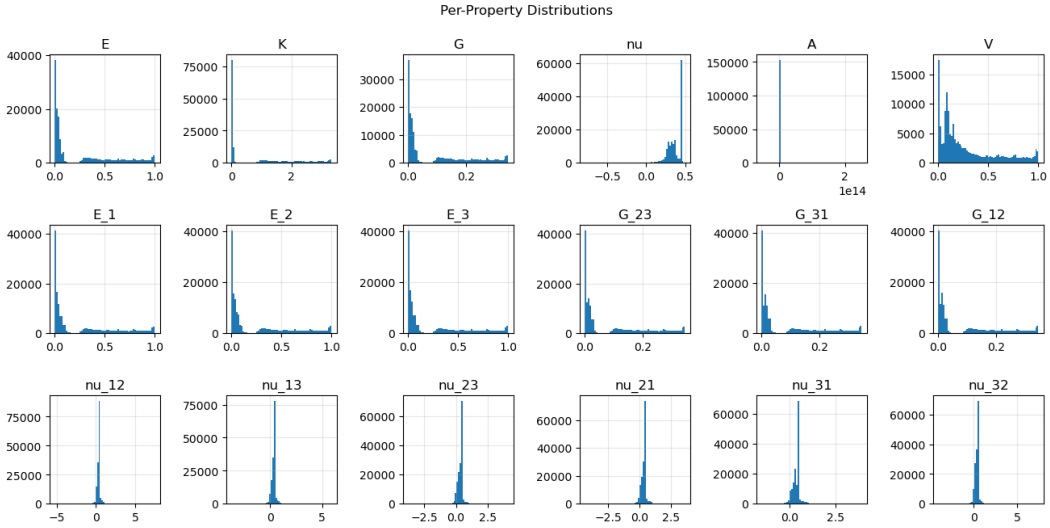


Figure 12: MetaDB material property distributions.

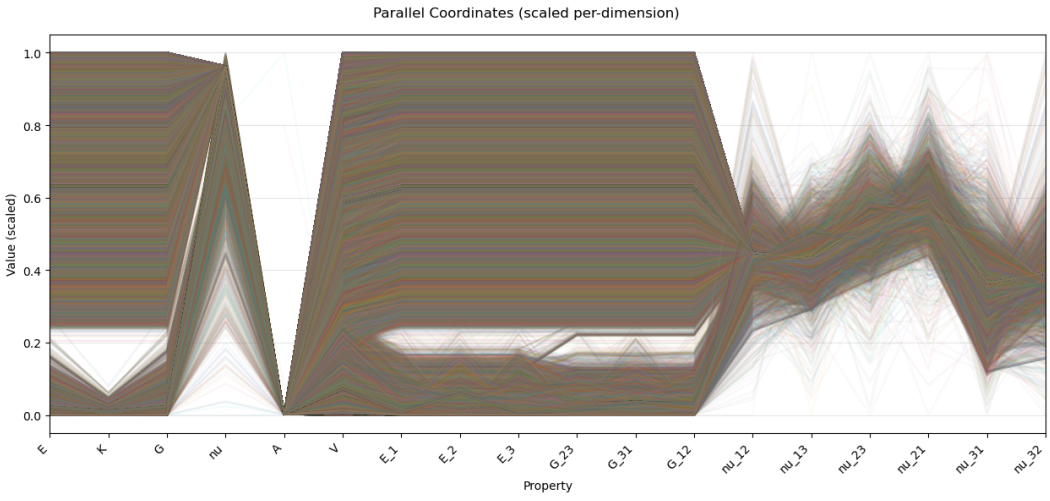


Figure 13: Properties of all MetaDB materials: each horizontal polyline is one material.

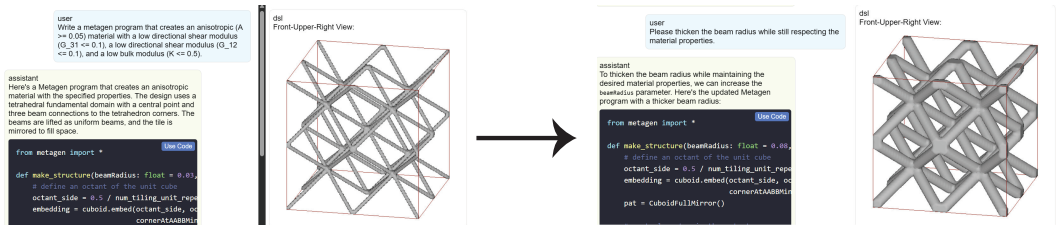


Figure 14: Iterative Inverse Design: Designers can specify desired target properties, and these preferences and constraints can be considered throughout multiple design iterations.

Table 3: Multi-task vs Single Task Training. **Bold** indicates significance (pair t-test, $p < .05$). Across tasks and models, multi-task training is almost always beneficial.

Category	Inverse Design		Material Understanding	Reconstruction		
Metric	Error ↓	Valid ↑	Error ↓	CD ↓	IoU ↑	Valid ↑
Model						
LLaVA	**.022 ± .004**	98.3%	.033 ± .006	**.031 ± .003**	**.505 ± .029**	94.2%
LLaVASingle	.041 ± .005	**100%**	.035 ± .005	.058 ± .003	.180 ± .017	91.8%
Nova	.018 ± .004	**88.5%**	.017 ± .003	**.034 ± .003**	**.463 ± .029**	97.8%
NovaSingle	.021 ± .005	84.2%	.024 ± .005	.040 ± .003	.389 ± .028	93.6%

verified this with our simulator). But design is always iterative, and seeing one design can spark new criteria and objectives. In this case we wanted a thicker structure that still conformed to our original input, and (again verified by simulation), the model was able to update the design within target parameters. This illustrates the powerful ability of language models to remember and carry through *design context*, allowing for assistance across multiple design iterations.

Both case studies were performed on an earlier version of MetaAssist-Nova with strictly worse performance the the currently benchmarked version.

F Further Benchmark Results

F.1 Ablations

F.2 Category Sub-Task Results

Tables 5, 6, and 7 break down Table 1 for each task category into its task variations (number of views, targets, etc.). These allow a more nuanced view of MetaAssist's capabilities.

In reconstruction (Table 5), we see that having more viewpoints generally improves reconstruction accuracy, though this tops out for LLaVA at 4 viewpoints. We also see that multiple viewpoints greatly improves o3's validity. In practice, this is because it frequently produce 2D structures when only given a single side-view, which do not meet our periodicity requirements for validity.

For the inverse design tasks in Table 6, there is a slight trend that an intermediate number of targets is easier than very few or very many. Our hypothesis is that with a small number of targets it is possible that all targets are correlated (e.g. elastic moduli), but this is eventually counteracted by having more targets to hit. More in-depth study is required to deduce why this happens.

Table 4: Effect of sampling temperature on benchmark scores. It has no significant effect benchmark performance. Testing was done on an earlier variant of the Nova model.

Category	Inverse Design		Material Understanding	Reconstruction		
Metric	Error ↓	Valid ↑	Error ↓	CD ↓	IoU ↑	Valid ↑
Temperature						
0.00001	.028 ± .003	**92.7%**	**.030 ± .004**	.045 ± .001	**.334 ± .007**	86.7%
0.7	**.026 ± .002**	91.4%	.032 ± .005	**.045 ± .001**	.334 ± .007	**87.2%**
Random	.028 ± .003	91.9%	.031 ± .004	.045 ± .001	.330 ± .007	87.2%

Table 5: Reconstruction Results Broken Down by task type.

Task	1 View			2 View			3 View			4 View		
Metric	CD↓	IoU↑	Valid	CD↓	IoU↑	Valid	CD↓	IoU↑	Valid	CD↓	IoU↑	Valid
Model												
LLaVA	**0.035**	**0.456**	93.9%	**0.032**	**0.498**	94.2%	**0.029**	**0.519**	94.6%	**0.031**	**0.505**	94.2%
Nova	0.037	0.424	**97.7%**	0.035	0.454	**97.6%**	0.035	0.464	**97.2%**	0.034	0.463	**97.8%**
NovaBase	0.119	0.049	18.7%	0.117	0.050	17.0%	0.118	0.053	22.0%	0.125	0.050	25.0%
OpenAIO3	0.052	0.150	36.8%	0.055	0.141	58.9%	0.052	0.151	62.6%	0.052	0.155	68.5%

Table 6: Inverse Design Results broken down by task type.

Task	1 Target		2 Target		3 Target		4 Target		5 Target		6 Target	
Metric	Error↓	Valid	Error↓	Valid	Error↓	Valid	Error↓	Valid	Error↓	Valid	Error↓	Valid
Model												
LLaVA	0.004	**100%**	0.024	**100%**	0.021	**99.0%**	0.022	**98.3%**	0.020	**97.3%**	0.021	**97.7%**
Nova	0.020	90.0%	**0.019**	92.8%	**0.016**	88.0%	**0.018**	88.5%	**0.017**	88.5%	**0.019**	91.1%
NovaBase	—	0.0%	0.164	3.0%	0.044	4.2%	0.042	2.1%	0.034	2.8%	0.073	2.3%
OpenAIO3	**0.000**	35.0%	0.033	44.7%	0.025	39.1%	0.031	35.7%	0.024	37.7%	0.031	35.8%

The expanded material understanding results shown in Table 7 reveals a difference between Nova and LLaVA; Nova is more able to take advantage of the extra image and code information, whereas LLaVA does best with only a single image. This mirrors the degredation of LLaVA in reconstruction at 4 images.

F.3 Result Galleries

We also present randomly[1] sampled queries for each task, and visualize their results across models, along with their benchmark metrics. This shows the qualitative differences between the models' performances, while grounding the numeric metrics to make them more understandable.

Figure 15 illustrates reconstruction from 4 viewpoint renders. Of particular interest is the o3 column on the far right. For 4/5 examples, o3 correctly reproduced the basic shape of the side-on views up-to the number of repeats. This suggests that it can correctly build skeletons, but struggles with selecting the correct embedding scale.

Figure 16 illustrates material prediction based on specified property requirements. In these examples, the LLaVA models successfully generate materials that meet the given criteria, but other models occasionally generate invalid materials or fail to satisfy the specified requirements.

Figure 17 illustrates generated materials' predicted versus actual properties. In these examples the LLaVA and OmniTask Nova models do quite well, but single task Nova and untuned models (Novalite and o3) fall behind.

[1] rejection filtered so that all models had valid outputs for the input, except for inverse design where this was not possible

Table 7: Material Understanding results broken down by task type.

Task	1 View	4 View + Code
Metric	Error ↓	Error ↓
Model		
LLaVA	0.028	0.033
Nova	**0.024**	**0.017**
NovaBase	0.206	0.192
OpenAIO3	0.083	0.071

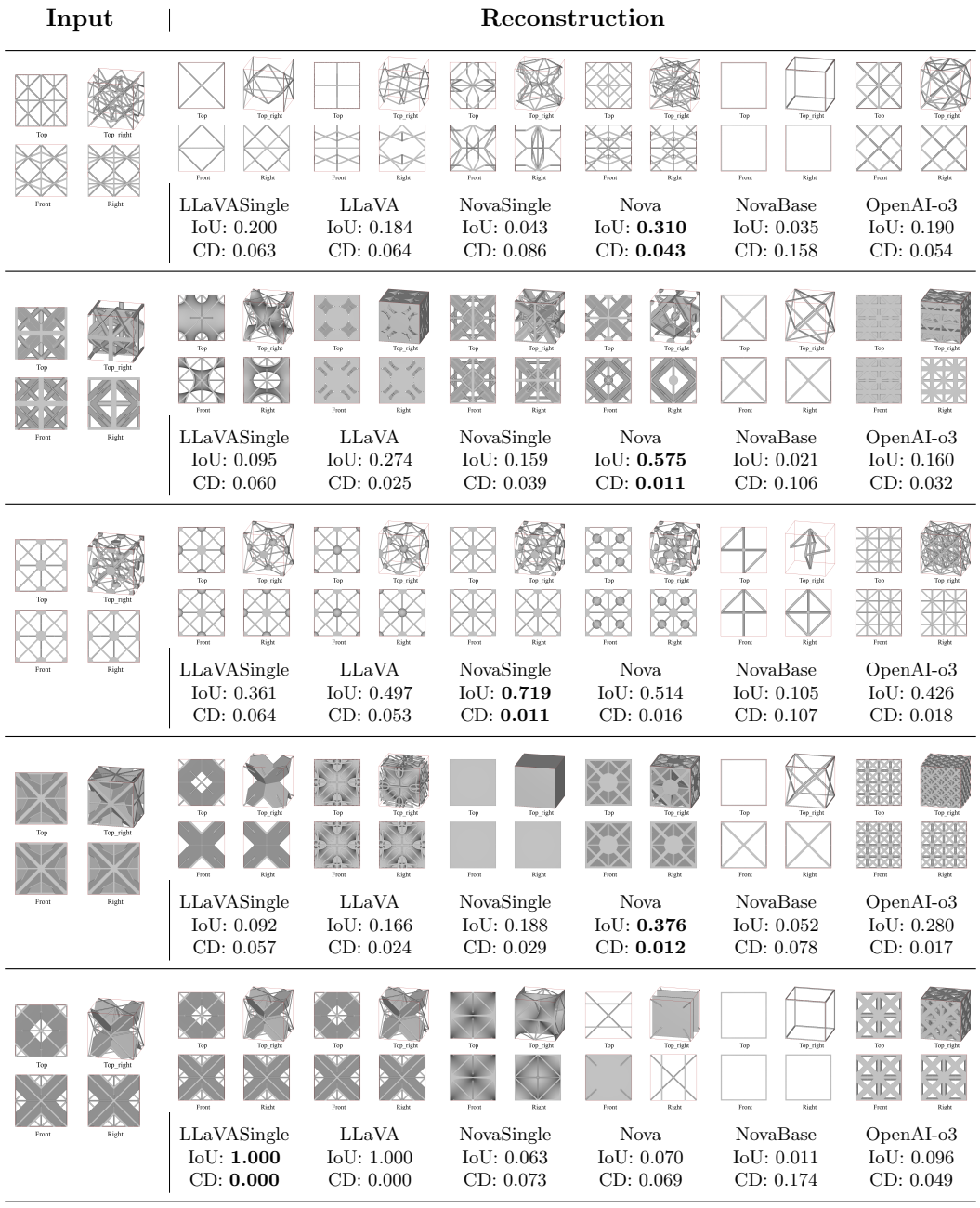


Figure 15: 4 View reconstruction results for random test samples by model. Left: the input renders shown to each model. Right: renders of predicted reconstructions.

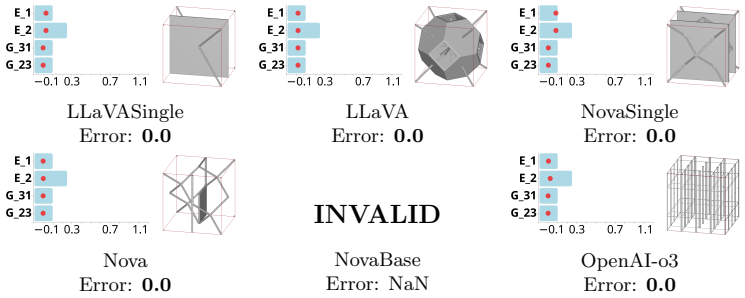
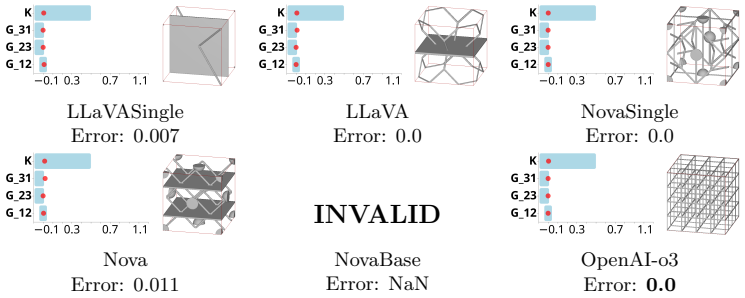
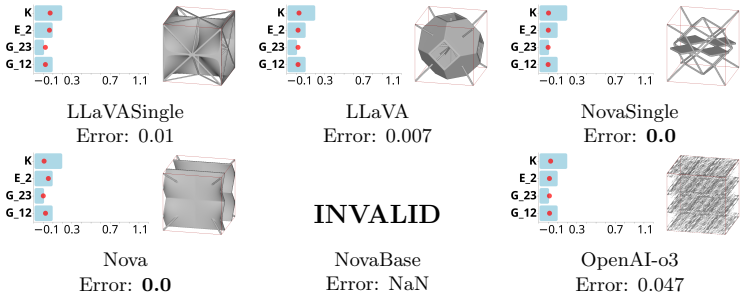
Write a metagen program that creates...	**Generated Design**
a material with a very low Young's modulus along the x direction ($\mathbf{E_1 \leq 0.1}$), a low directional shear modulus ($\mathbf{G\_31 \leq 0.1}$), a low directional shear modulus ($\mathbf{G\_23 \leq 0.1}$), and a low Young's modulus along the y direction ($\mathbf{E\_2 \leq 0.25}$).	 LLaVASingle Error: **0.0** LLaVA Error: **0.0** NovaSingle Error: **0.0** Nova Error: **0.0** **INVALID** NovaBase Error: NaN OpenAI-o3 Error: **0.0**
a compressible ($\mathbf{K \leq 0.5}$) material with a very low directional shear modulus ($\mathbf{G\_23 \leq 0.01}$), a directional shear modulus ($\mathbf{G\_12}$) of roughly 3e-05, and a very low directional shear modulus ($\mathbf{G\_31 \leq 0.01}$).	 LLaVASingle Error: 0.007 LLaVA Error: 0.0 NovaSingle Error: 0.0 Nova Error: 0.011 **INVALID** NovaBase Error: NaN OpenAI-o3 Error: **0.0**
a material with a very low Young's modulus along the vertical direction ($\mathbf{E\_2 \leq 0.1}$), a very low directional shear modulus ($\mathbf{G\_23 \leq 0.01}$), a low directional shear modulus ($\mathbf{G\_12 \leq 0.1}$), and a very low bulk modulus ($\mathbf{K \leq 0.2}$).	 LLaVASingle Error: 0.01 LLaVA Error: 0.007 NovaSingle Error: **0.0** Nova Error: **0.0** **INVALID** NovaBase Error: NaN OpenAI-o3 Error: 0.047

Figure 16: Inverse design results for a random selection of queries. Left: the text query given to each model. Right: paired data showing – for each model – an image of the generated structure alongside a property profile comparison. This profile shows the target values/ranges (in blue), versus simulated properties of the predicted materials (in red). Red arrows indicate that the predicted value is beyond the chart boundaries. Some models failed to produce a valid model for certain queries, indicated by the label "INVALID".

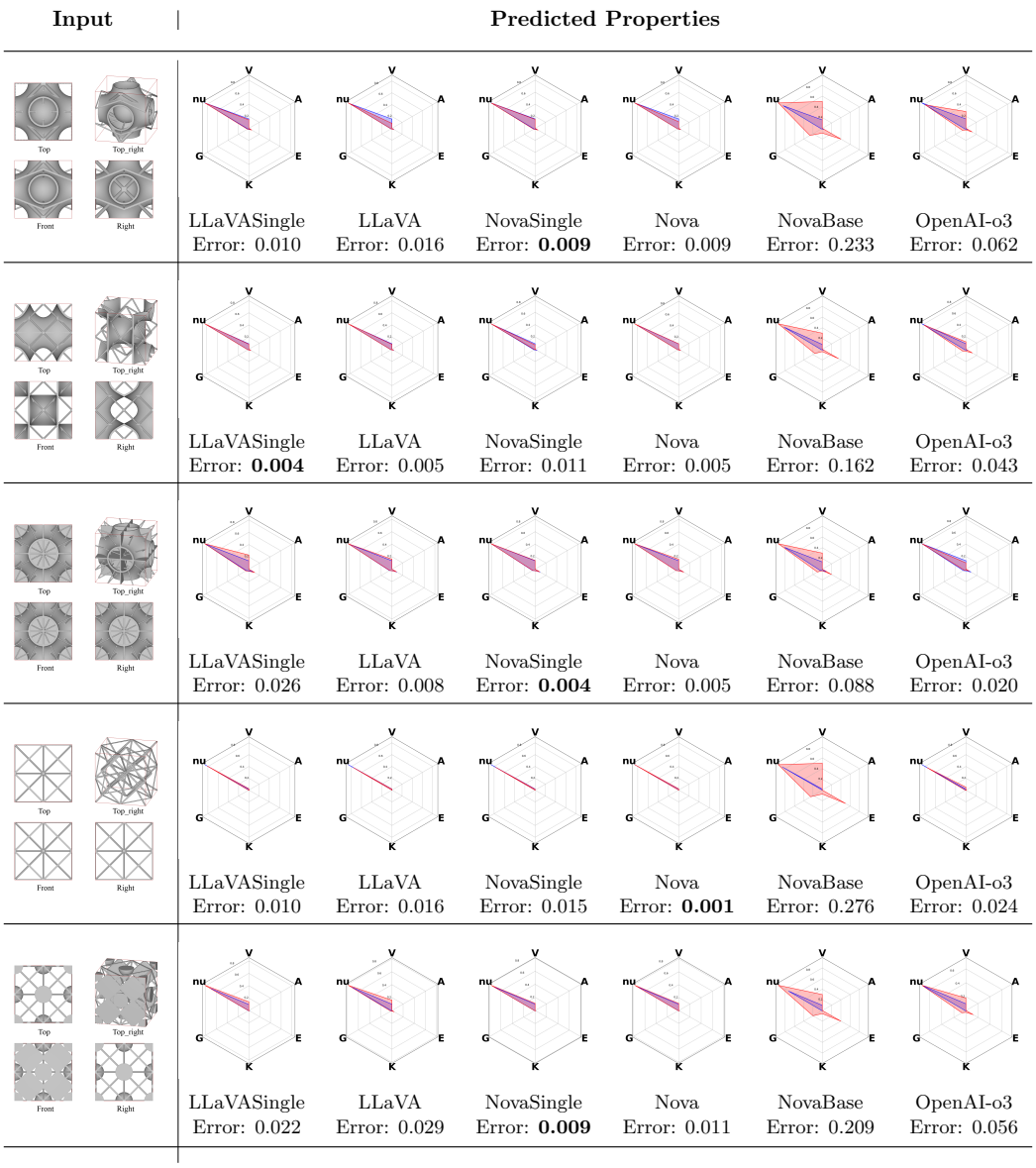


Figure 17: Material property predictions given 4 input views (shown) and the program code (not shown). The radar charts plot the 6 averaged property values (scaled and shifted to always be positive). The blue regions show the ground truth values, while red shows the prediction.

G METABENCH

G.1 INTERMEDIATE REPRESENTATION

Each dataset is given by a set of .jsonl files: one file each for train, validate, and test. Each line of a .jsonl file describes a single example using a dictionary with the following keys:

- **'task_type'**: a string identifying the task category; in our case, it is one of {'reconstruction', 'inverse_design', 'material_understanding'}.
- **'label'**: unique text label identifying this task entry, using descriptive elements where applicable, such as provided image viewpoints or source files.
- **'source'**: [if applicable] path to the source metamaterial, relative to the database root (and including the leading '/')
- **'data'**: any and all data required to run evaluations, including references for large elements (e.g. images, meshes, etc.) and/or directly embedded values.
- **'query'**: natural language framing of the question to be provided to an LLM. Any images (or other non-text input) must be specified by reference.
- **'response'**: [optional] an expected response from an LLM that has been asked 'query'. This field is permitted to exist for a test example; removal of this information is the responsibility of the LLM-specific formatters, when required.

The system prompt has been purposefully excluded, both because it would be very large, and because that is an implementation detail of a predictive model, and not part of the benchmark itself.

G.2 TASK CONSTRUCTION FOR INVERSE DESIGN

Inverse design tasks are specified as a collection of target values or bounded-ranges for a subset of material properties, from which we construct a natural-language query that describes that set of targets. Creating these tasks has two stages: selecting a set of targets, and generating an grammatically correct English sentence from those targets.

Property References To aid in this process, we generate a reference dictionary with information about each of the 18 properties, of the following form:

```
{
'nu': {
    "full_prop_name": "Poisson ratio",
    "alternate_symbols": ["nu_{VRH}"],
    "property_generality": PropertyGenerality.OVERALL,
    "property_type": PropertyType.POISSON_RATIO,
    "dataset_coverage": {
        "min": -0.5,
        "max": 0.5,
        "q1": 0.3,
        "q3": 0.36,
        "densely_populated_ranges": [[0.2, 0.4]]
    },
    "smallest_meaningful_quantization": 0.01,
    "adjective_descriptors":[{"description": f"auxetic", "target_type": TargetType.
        UPPER_BOUND, "target_value":0}],
    "property_descriptors": [{"description": f"a negative Poisson ratio", "target_type":
        TargetType.UPPER_BOUND, "target_value":0},
                             {"description": f"a positive Poisson ratio", "target_type":
                                 TargetType.LOWER_BOUND, "target_value":0}],
    "verb_descriptors":      [{"description": f"contracts transversely under axial compression", "
        target_type": TargetType.UPPER_BOUND, "target_value":0},
                             {"description": f"expands transversely under axial compression", "
                                 target_type": TargetType.LOWER_BOUND, "target_value":0},
                             {"description": f"contracts in other directions when compressed
                                 along one axis", "target_type": TargetType.UPPER_BOUND, "
                                 target_value":0},
```

```
                {"description": f"expands in other directions when compressed along one axis", "target_type": TargetType.LOWER_BOUND, "target_value":0},
                {"description": f"expands transversely under axial elongation", "target_type": TargetType.UPPER_BOUND, "target_value":0},
                {"description": f"contracts transversely under axial elongation", "target_type": TargetType.LOWER_BOUND, "target_value":0},
                {"description": f"expands in other directions when stretched along one axis", "target_type": TargetType.UPPER_BOUND, "target_value":0},
                {"description": f"contracts in other directions when stretched along one axis", "target_type": TargetType.LOWER_BOUND, "target_value":0}]
},
}
```

The full listing for all 18 properties is available in the metagen code provided in the supplement: `metagen/benchmarks_inverse_design.py`.

These entries provide information about the property ranges, dataset coverage, and interesting value breakpoints together with phrases that might be used to request them (e.g., "auxetic" implies $\nu < 0$). All aspects of these reference entries will be used in the following subsections to construct robust, varied and meaningful property queries for different material examples.

Active Property Selection For a given structure, we enforce that the "active" property subset follows two rules. First, the active set may only employ the overall values *or* the directional values for any given property – e.g., if a profile includes measure(s) for Young's modulus, it may either include the overall Young's modulus E *or* one or more of the directional values $\{E_1, E_2, E_3\}$; however, it is not permitted to simultaneously include E and one or more directional variants. Moreover, a profile is only allowed to use directional variants if it is sufficiently anisiotropic. We chose our anisotropy threshold as $A \geq 0.0025$, based on a manual exploration of the correlation between material spheres and anisotropy values appearing in our dataset. Subject to these rules, we select the "active" subset of properties based on a heuristic that determines the most interesting or salient properties of a given model.

We construct this heuristic score by examining individual properties of a model, and assigning a reward or penalty based on the expected notability of a particular characteristic or combination thereof. For example, if a material is near isotropic ($A < 0.0025$), we strongly reward the anisotropy property (so it is likely to end up in the active set) and heavily penalize all directional properties (so they will not be activated, as they are not likely to be notable). If the material is sufficiently anisotropic, we look at each property with directional variants, then compute pairwise differences between the values (e.g. E_1 vs. E_2). The directional properties are rewarded proportionally to each pairwise difference, so directions with larger discrepancies are more likely to be activated. Independently, we examine the ratio between the Young's modulus E and the volume fraction V – if the ratio is high (i.e., the material preserves stiffness with dramatically less material / lighter weight, which is a highly sought after combination), we strongly reward both properties. Finally, we examine each property in turn, and award additional points if they exhibit values that are extreme and/or underrepresented in our dataset. The reward is proportional to the relative extremity and inversely proportional to representation.

Given these scores, we iteratively select the highest-reward properties that preserve our overall active set rules. To ensure some variation in our inverse design profiles, we also introduce the opportunity to add randomly chosen properties into our profile: after each active set addition from the ranked data, we break the loop with some low probability (10%) and fill the remaining slots with randomly chosen properties that respect the rules relative to our partial active set.

Active Property Target Selection For each active property, we must now select a target value or range. To do this, we evaluate the options present in our reference dictionary, and extract all targets that are satisfied by the material at hand. We organize these into groups based on value and target type (range, value, lower/upper bound). Then, we choose the group that offers the tightest bound relative to the current material's property value. If multiple bound types are associated with the chosen target

value, we select a bound type at random. Finally, we construct a profile with all targets matching the selected value and bound type. Assuming an example material where the Poisson ratio $\nu = -0.1$, the resulting profile might be as follows:

```
{
    "property": "nu"
    " target_value ": 0
    " target_type ": "upper_bound"
    "  target_descriptions  ": [
        {
            " description ": "auxetic",
            " description_type ": " adjective "
        },
        {
            " description ": "a  negative  Poisson  ratio ",
            " description_type ": "noun"
        },
        {
            " description " : " contracts  transversely  under  axial  compression",
            " description_type ": "verb"
        },
        {
            " description " : " contracts  in  other  directions  when compressed along one  axis",
            " description_type ": "verb"
        },
        {
            " description " : "expands  transversely  under  axial  elongation",
            " description_type ": "verb"
        },
        {
            " description " : "expands in  other  directions  when stretched  along one  axis",
            " description_type ": "verb"
        }
    ]
}
```

Query Construction We want to create varied sentence structures to train and test against. To do this, each target type (value, upper bound, or lower bound) and target property has associated with it several descriptive phrases, as shown in the profile above. These phrases are paired with a part of speech (adjective, noun, or verb). As examples "very dense" (adjective), "contracts in the X direction when the Y direction is stretched" (verb), or "a negative Poisson ratio in at least one direction" (noun). Phrases that do not include numeric targets are accompanied by a parenthetical aside given a target value or range (e.g. "very dense ($V > 0.8$)."

We start by randomly selecting one phrase for each target property, binning them by part of speech, then randomizing the order within bins. Adjectives are further randomly split between *front-adjectives* that precede the noun "material" ("a very dense material") and *back-adjectives* that follow it ("a material that is very dense"). We then form a query string by applying the template:

```
Write a metagen program that creates [a/an] { front_adjectives } material { back_adjectives } {verbs}
    {nouns}.
```

The template strings are augmented with part-of-speech appropriate connectors ("that is", "with", "that", "and"), and commas, depending on the parts of number of each part of speech in each position. The pronoun (a/an) as selected based on the first letter of `{front_adjectives}` if there are any, otherwise "a" for "a material".

H Implementation Details

LLaVA LLaVA and LLaVASingle tune Llama3-LLaVA-Next-8b Li et al. (2024); Liu et al. (2024) using low-rank adaptation Hu et al. (2022), with with $r = 16$ and $\alpha = 32$. Models were optimized using AdamW Loshchilov & Hutter (2017) with a 1e-5 learning rate and a cosine learning rate

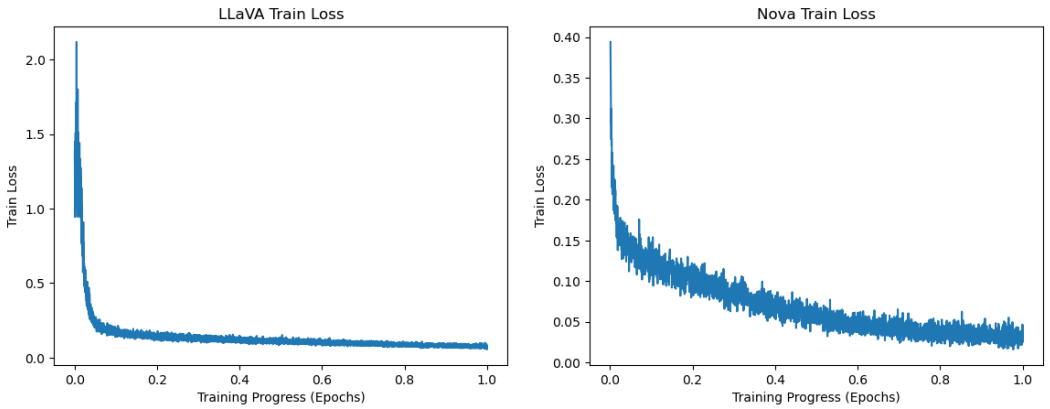


Figure 18: Training loss for LLaVA and Nova models. The smaller LLaVA model converges with significantly fewer examples than the larger Nova model.

scheduler with 0.03 warm-up ratio. Single models were trained on for 1 epoch on 4 NVIDIA B100 GPUs for 2 hours each with a batch size of 16, while the generalist LLaVA model was trained for 1 epoch on 8 B200 GPUs with a batch size of 32 over 25 hours due to its significantly larger training set, and for parity with the genaralist Nova model. During inference, the temperature was set to 0 to ensure deterministic outputs. We trained LLaVA on Amazon EC2 p6-b200.48xlarge instances. Our distributed training implementation achieved performance metrics with peak GPU utilization reaching 99% per GPU and peak memory utilization at approximately 95% per GPU across the B200s' 180GB HBM3 capacity.

Nova We implemented Nova full-rank supervised fine-tuning on the Nova Lite architecture (amazon.nova-lite-v1:0:300k) using Amazon SageMaker distributed training infrastructure with ml.p5.48xlarge instances, each equipped with 8 NVIDIA H100 GPUs featuring 80GB HBM3 memory. Our training configuration employed a maximum sequence length of 32,768 tokens with a global batch size of 64, utilizing the distributed fused Adam optimizer in AdamW mode with a learning rate of 5×10^{-6}, beta parameters of (0.9, 0.999), epsilon of 1×10^{-6}, and zero weight decay. The learning rate schedule incorporated 10 warmup steps followed by decay to 1×10^{-6}, while all dropout mechanisms (hidden, attention, and feed-forward network) were disabled to perform full-rank fine-tuning across all model parameters. All models were trained for 1 epoch, where the multi-domain task required approximately 15 hours using 16 P5 instances. Our distributed training implementation achieved notable performance metrics with peak GPU utilization reaching 95% per GPU, sustained utilization of 75-95% per GPU during core training phases, and stable memory utilization at approximately 43% per GPU across the H100s' 80GB HBM3 capacity.

H.1 Training Curves

Both the LLaVA and Nova models were trained for 1 epoch. Figure 18 shows the training curves for each model, demonstrating that the LLaVA model had converged more and more quickly than the Nova model.

H.2 Timing and Costs

MetaDSL execution and simulation time dominate LLM inference time for material generation. These are highly variable based on the geometric complexity of the generated program, with the majority executing and simulating in 5 minutes or less. MetaAssist generations are on average more time-complex that MetaDB (see Table 8. In practice, MetaAssist latencies are much lower because we do not run simulations in the interactive system.

Since MetaDSL is quite compact, inference can be performed efficiently with few tokens. The majority of the inference tokens are taken by the common API-description system prompt (Section I.1), the cost of which can be amortized by caching. Using NovaOmni (ignoring caching for simplicity), the

Program Source	Avg. (s)	Median (s)	Std (s)
MetaDB	181	123	328
MetaAssist	591	290	746

Table 8: MetaDSL Execution and simulation times for program in MetaDB, and programs generated by MetaAssist-Nova over the MetaBench test set (reconstruction and inverse design).

average MetaBench query used 8730 tokens (8284 input and 446 output). At current API pricing, the average query would cost $0.0006, and inference for the full test set would cost $7.11.

I Query Templates

For training models and running inference, we used prompt templates and inserted details for each specific query. In the following templates, <[...]> is used as a delimiter to denote the inclusion of an image.

I.1 Universal System Prompt

For consistency, every example was provided with a common system prompt that describes the Metagen DSL, explains the material properties and rendered views we have in our dataset, and describes the basic task categories.

```
You are an expert metamaterials assistant that generates and analyzes cellular metamaterial
    designs based on material properties, images, and programatic definitions in the Metagen
    metamaterial DSL.


# Procedural Description in a Metamaterial DSL:

{ api_description }

# Material Analysis:
You can analyze the density, anisotropy, and elasticity properties of metamaterials. All
    metamaterials are assumed to be constucted from an isotropic base material with Poisson's
    ratio nu = 0.45.
The Young's Modulus of this base material is not specified, instead, the elastic moduli of the
    metamaterials -- Young's Modulus (E), Bulk Modulus (K), and Shear Modulus (G), are expressed
    relative to the base material Young's modulus (E_base). This means, for example, that
    relative Young's Moduli can range from 0 to 1. The material properties you can analyze are:

- E: Young's Modulus, Voigt-Reuss-Hill (VRH) average, relative to E_base
- E_1,E_2,E_3: Directional Young's Moduli, relative to E_base
- G: Shear Modulus (VRH average), relative to E_base
- G_23,G_13,G_12: Directional Shear Moduli, relative to E_base
- nu: Poisson ratio (VRH average)
- nu_12, nu_13, nu_23, nu_21, nu_31, nu_32: Directional Poisson ratios
- K: Bulk modulus (VRH average), relative to E_base
- A: Anisotropy (universal anisotropy index)
- V: Volume Fraction

# Material Images:

Images of metamaterials depict a base cell of the material rendered from four viewpoints:

- from the top
- from the front side
- from the right side
- from an angle at the upper-front-right

# Tasks:
```

```
You will be asked to perform several kinds of tasks:

- Reconstruction: from one or more images of a target material, reconstruct a Metagen program that
    generates the metamaterial in the images.
- Inverse Design: from a description of the properties of a desired materials, write a Metagen
    program that creates a metamaterial with those properties.
- Material Understanding: from images of a metamaterial and/or a Metagen program, analyze a
    material and predict its properties.
```

I.2 METADSL API

The Metagen language description (inserted as the `api_description` in the system prompt above) is as follows:

```
Programs in Metagen are built in two stages: one that creates local geometric structure, and a
    second that patterns this structure throughout space. Each of these is further broken down
    into subparts.


==================================
    API description (Boilerplate)
==================================
Each program is given as a python file (.py).
This program must import the metagen package and define a function called "make_structure()",
    which returns the final Structure object defined by the program.
If parameters are present in make_structure(), they MUST have a default value.
Specifically, the file structure is as follows:


from metagen import *

def make_structure(...) -> Structure:
    <content>



==================================
    DSL description
==================================

======= Skeleton Creation ========
vertex(cpEntity, t)
    @description:
        Create a new vertex. This vertex is defined relative to its containing convex polytope (
            CP). It will only have an embedding in R3 once the CP has been embedded.
    @params:
        cpEntity    - an entity of a convex polytope (CP), referenced by the entity names.
        t           - [OPTIONAL] list of floats in range [0,1], used to interpolate to a specific
            position on the cpEntity.
                      If cpEntity is a corner, t is ignored.
                      If cpEntity is an edge, t must contain exactly 1 value. t is used for
                          linear interpolation between the endpoints of cpEntity.
                      If cpEntity is a face, t must contain exactly 2 values. If cpEntity is a
                          triangular face, t is used to interpolate via barycentric coordinates
                          . If cpEntity is a quad face, bilinear interpolation is used.

                      If the optional interpolant t is omitted for a non-corner entity, the
                          returned point will be at the midpoint (for edge) or the centroid (
                          for face) of the entity. Semantically, we encourage that t be
                          excluded (1) if the structure would be invalid given a different non
                          -midpoint t, or (2) if the structure would remain unchanged in the
                          presence a different t (e.g., in the case of a conjugate TPMS,
                          where only the entity selection matters).
```

```
    @returns:
        vertex      - the new vertex object
    @example_usage:
        v0 = vertex(cuboid.edges.BACK_RIGHT, [0.5])
        v1 = vertex(cuboid.edges.TOP_LEFT)


Polyline(ordered_verts)
    @description:
        Creates a piecewise-linear path along the ordered input vertices. All vertices must be
            referenced to the same CP (e.g., all relative to cuboid entities). The resulting path
            will remain a polyline in any structures that include it.
    @params:
        ordered_verts   - a list of vertices, in the order you'd like them to be traversed. A
            closed loop may be created by repeating the zeroth element at the end of the list.
            No other vertex may be repeated. Only simple paths are permitted.
    @returns:
        polyline        - the new polyline object
    @example_usage:
        p0 = Polyline([v2, v3])
        p0 = Polyline([v0, v1, v2, v3, v4, v5, v0])


Curve(ordered_verts)
    @description:
        Creates a path along the ordered input vertices. This path will be smoothed at a later
            stage (e.g., to a Bezier curve), depending on the lifting procedures that are chosen.
            All input vertices must be referenced to the same CP (e.g., all relative to cuboid
            entities).
    @params:
        ordered_verts   - a list of vertices, in the order you'd like them to be traversed. A
            closed loop may be created by repeating the zeroth element at the end of the list.
            No other vertex may be repeated. Only simple paths are permitted.
    @returns:
        curve           - the new curve object
    @example_usage:
        c0 = Curve([v2, v3])
        c0 = Curve([v0, v1, v2, v3, v4, v5, v0])

skeleton(entities)
    @description:
        Combines a set of vertices OR polylines/curves into a larger structure, over which
            additional information can be inferred. For example, within a skeleton, multiple
            open polylines/curves may string together to create a closed loop, a branched path,
            or a set of disconnected components.
    @params:
        entities        - a list of entities (vertices or polylines/curves) to be combined. A
            given skeleton must only have entities with the same dimension -- that is, it must
            consist of all points or all polylines/curves.
    @returns:
        skeleton        - the new skeleton object
    @example_usage:
        skel = skeleton([curve0, polyline1, curve2, polyline3])
        skel = skeleton([v0])


======= Lifting Procedures ========
UniformBeams(skel, thickness)
    @description:
        Procedure to lift the input skeleton to a 3D volumetric structure by instantiating a beam
            of the given thickness centered along each polyline/curve of the input skeleton.
    @requirements:
        The skeleton must contain only polylines and/or curves. The skeleton must not contain any
            standalone vertices.
    @params:
```

```
        skel            - the skeleton to lift
        thickness       - the diameter of the beams
    @returns:
        liftProc        - the lifted skeleton
    @example_usage:
        liftProcedure = UniformBeams(skel, 0.03)

SpatiallyVaryingBeams(skel, thicknessProfile)
    @description:
        Procedure to lift the input skeleton to a 3D volumetric structure by instantiating a beam
            of the given spatially-varying thickness profile centered along each polyline/curve
            of the input skeleton.
    @requirements:
        The skeleton must contain only polylines and/or curves. The skeleton must not contain any
            standalone vertices.
    @params:
        skel            - the skeleton to lift
        thicknessProfile - specifications for the diameter of the beams along each polyline/curve.
            Given as a list[list[floats]], where the each of the n inner lists gives the
            information for a single sample point along the polyline/curve. The first element in
            each inner list provides a position parameter t\\in[0,1] along the polyline/curve,
            and the second element specifies the thickness of the beam at position t
    @returns:
        liftProc        - the lifted skeleton
    @example_usage:
        liftProcedure = SpatiallyVaryingBeams(skel, 0.03)

UniformDirectShell(skel, thickness)
    @description:
        Procedure to lift the input skeleton to a 3D volumetric structure by inferring a surface
            that conforms to the boundary provided by the input skeleton. The surface is given by
            a simple thin shell model: the resulting surface is incident on the provided
            boundary while minimizing a weighted sum of bending and stretching energies. The
            boundary is fixed, though it may be constructed with a mix of polylines and curves (
            which are first interpolated into a spline, then fixed as part of the boundary). The
            skeleton must contain a single closed loop composed of one or more polylines and/or
            curves. The skeleton must not contain any standalone vertices.
    @requirements:

    @params:
        skel            - the skeleton to lift
        thickness       - the thickness of the shell. The final offset is thickness/2 to each side
            of the inferred surface.
    @returns:
        liftProc        - the lifted skeleton
    @example_usage:
        liftProcedure = UniformDirectShell(skel, 0.1)

UniformTPMSShellViaConjugation(skel, thickness)
    @description:
        Procedure to lift the input skeleton to a 3D volumetric structure by inferring a triply
            periodic minimal surface (TPMS) that conforms to the boundary constraints provided by
            the input skeleton. The surface is computed via the conjugate surface construction
            method.
    @requirements:
        The skeleton must contain a single closed loop composed of one or more polylines and/or
            curves. The skeleton must not contain any standalone vertices.
        Each vertex in the polylines/curves must live on a CP edge.
        Adjacent vertices must have a shared face.
        The loop must touch every face of the CP at least once.
        If the CP has N faces, the loop must contain at least N vertices.
    @params:
        skel            - the skeleton to lift
        thickness       - the thickness of the shell. The final offset is thickness/2 to each side
            of the inferred surface.
```

```
    @returns:
        liftProc          - the lifted skeleton
    @example_usage:
        liftProcedure = UniformTPMSShellViaConjugation(skel, 0.03)

UniformTPMSShellViaMixedMinimal(skel, thickness)
    @description:
        Procedure to lift the input skeleton to a 3D volumetric structure by inferring a triply
            periodic minimal surface (TPMS) that conforms to the boundary constraints provided by
            the input skeleton. The surface is computed via mean curvature flow. All polyline
            boundary regions are considered fixed, but any curved regions may slide within their
            respective planes in order to reduce surface curvature during the solve.
    @requirements:
        The skeleton must contain a single closed loop composed of one or more polylines and/or
            curves. The skeleton must not contain any standalone vertices.
        Each vertex in the polylines/curves must live on a CP edge.
        Adjacent vertices must have a shared face.
    @params:
        skel              - the skeleton to lift
        thickness         - the thickness of the shell. The final offset is thickness/2 to each side
            of the inferred surface.
    @returns:
        liftProc          - the lifted skeleton
    @example_usage:
        liftProcedure = UniformTPMSShellViaMixedMinimal(skel, 0.03)

Spheres(skel, thickness)
    @description:
        Procedure to lift the input skeleton to a 3D volumetric structure by instantiating a
            sphere of the given radius centered at vertex p, for each vertex in the skeleton.
    @requirements:
        The skeleton must only contain standalone vertices; no polylines or curves can be used.
    @params:
        skel              - the skeleton to lift
        thickness         - the sphere radius
    @returns:
        liftProc          - the lifted skeleton
    @example_usage:
        s_lift = Spheres(skel, 0.25)


======= Tile Creation ========
Tile(lifted_skeletons, embedding)
    @description:
        Procedure to embed a copy of the skeleton in R^3 using the provided embedding information.
            The embedding information can be computed by calling the "embed" method of the
            relevant CP.
    @requirements:
        The embedding information must correspond to the same CP against which the vertices were
            defined. For example, if the vertices are defined relative to the cuboid, you must
            use the cuboid.embed() method.
    @params:
        lifted_skeletons - a list of lifted skeleton entities to embed in R^3. All entities must
            reside in the same CP type, and this type must have N corners.
        embedding         - information about how to embed the CP and its relative skeletons within
            R^3. Obtained using the CP's embed() method
    @returns:
        tile              - the new tile object
    @example_usage:
        embedding = cuboid.embed(side_len, side_len, side_len, cornerAtAABBMin=cuboid.corners.
            FRONT_BOTTOM_LEFT)
        s_tile = Tile([beams, shell], embedding)


======= Patterning Procedures ========
```

```
TetFullMirror()
    @description:
        Procedure which uses only mirrors to duplicate a tet-based tile such that it partitions R
            ^3
    @params:
        N/A
    @returns:
        pat      - the patterning procedure
    @example_usage:
        pat = TetFullMirror()

TriPrismFullMirror()
    @description:
        Procedure which uses only mirrors to duplicate a triangular prism-based tile such that it
            partitions R^3
    @params:
        N/A
    @returns:
        pat      - the patterning procedure
    @example_usage:
        pat = TriPrismFullMirror()

CuboidFullMirror()
    @description:
        Procedure which uses only mirrors to duplicate an axis-aligned cuboid tile such that it
            fills a unit cube, such that it partitions R^3. Eligible cuboid CPs must be such
            that all dimensions are 1/(2^k) for some positive integer k.
    @params:
        N/A
    @returns:
        pat      - the patterning procedure
    @example_usage:
        pat = CuboidFullMirror()

Identity()
    @description:
        No-op patterning procedure.
    @params:
        N/A
    @returns:
        pat      - the patterning procedure
    @example_usage:
        pat = Identity()

Custom(patternOp)
    @description:
        Environment used to compose a custom patterning procedure. Currently only implemented for
            the Cuboid CP.
    @params:
        patternOp- outermost pattern operation in the composition
    @returns:
        pat      - the complete patterning procedure
    @example_usage:
        pat = Custom(Rotate180([cuboid.edges.BACK_RIGHT, cuboid.edges.BACK_LEFT], True,
                               Rotate180([cuboid.edges.TOP_RIGHT], True)))

Mirror(entity, doCopy, patternOp)
    @description:
        Pattern operation specifying a mirror over the provided CP entity, which must be a CP
            Face. Can only be used inside of a Custom patterning environment.
    @params:
        entity   - CP Face that serves as the mirror plane.
        doCopy   - boolean. When True, applies the operation to a copy of the input, such that the
            original and the transformed copy persist. When False, directly transforms the input
            .
```

```
        patternOp- [OPTIONAL] outermost pattern operation in the sub-composition, if any
    @returns:
        pat       - the composed patterning procedure, which may be used as is (within the Custom
                    environment), or as the input for further composition
    @example_usage:
        pat = Custom(Mirror(cuboid.faces.TOP, True,
                            Mirror(cuboid.faces.LEFT, True)))

Rotate180(entities, doCopy, patternOp)
    @description:
        Pattern operation specifying a 180 degree rotation about the provided CP entity. Can only
            be used inside of a Custom patterning environment.
    @params:
        entities  - List of CP entities, which define the axis about which to rotate. If a single
                    entity is provided, it must be a CP Edge. If multiple entities, they will be used to
                    define a new entity that spans them. For example, if you provide two corners, the
                    axis will go from one to the other. If you provide two CP Edges, the axis will reach
                    from the midpoint of one to the midpoint of the other.
        doCopy    - boolean. When True, applies the operation to a copy of the input, such that the
                    original and the transformed copy persist. When False, directly transforms the input
                    .
        patternOp- [OPTIONAL] outermost pattern operation in the sub-composition, if any
    @returns:
        pat       - the composed patterning procedure, which may be used as is (within the Custom
                    environment), or as the input for further composition
    @example_usage:
        pat = Custom(Rotate180([cuboid.edges.FRONT_LEFT, cuboid.edges.FRONT_RIGHT], True))

Translate(fromEntity, toEntity, doCopy, patternOp)
    @description:
        Pattern operation specifying a translation that effectively moves the fromEntity to the
            targetEntity. Can only be used inside of a Custom patterning environment.
    @params:
        fromEntity- CP Entity that serves as the origin of the translation vector. Currently only
                    implemented for a CP Face.
        toEntity  - CP Entity that serves as the target of the translation vector. Currently only
                    implemented for a CP Face.
        doCopy    - boolean. When True, applies the operation to a copy of the input, such that the
                    original and the transformed copy persist. When False, directly transforms the input
                    .
        patternOp- [OPTIONAL] outermost pattern operation in the sub-composition, if any
    @returns:
        pat       - the composed patterning procedure, which may be used as is (within the Custom
                    environment), or as the input for further composition
    @example_usage:
        gridPat = Custom(Translate(cuboid.faces.LEFT, cuboid.faces.RIGHT, True,
                                   Translate(cuboid.faces.FRONT, cuboid.faces.BACK, True)))


======= Structure Procedures ========
Structure(tile, pattern)
    @description:
        Combines local tile information (containing lifted skeletons) with the global patterning
            procedure to generate a complete metamaterial.
    @params:
        tile      - the tile object, which has (by construction) already been embedded in 3
                    D space, along with all lifted skeletons it contains.
        pattern   - the patterning sequence to apply to extend this tile throughout space
    @returns:
        structure - the new structure object
    @example_usage:
        obj = Structure(tile, pat)

Union(A, B)
    @description:
```

```
        Constructive solid geometry Boolean operation that computes the union of two input
            structures . The output of Union(A,B) is identical to Union(B,A)
    @params:
        A           - the first Structure to be unioned. This may be the output of Structure ,
            Union, Subtract , or Intersect
        B           - the second Structure to be unioned. This may be the output of Structure ,
            Union, Subtract , or Intersect
    @returns:
        structure   - the new structure object containing union(A,B)
    @example_usage:
        final_obj = Union(schwarzP_obj, Union(sphere_obj , beam_obj))

Subtract (A, B)
    @description:
        Constructive solid geometry Boolean operation that computes the difference (A - B) of two
            input structures . The relative input order is critical .
    @params:
        A           - the first Structure , from which B will be subtracted . This may be the
            output of Structure , Union, Subtract , or Intersect
        B           - the second Structure , to be subtracted from A. This may be the output of
            Structure , Union, Subtract , or Intersect
    @returns:
        structure   - the new structure object containing (A - B)
    @example_usage:
        final_obj = Subtract (c_obj , s_obj )

Intersect (A, B)
    @description:
        Constructive solid geometry Boolean operation that computes the intersection of two input
            structures , A and B.
    @params:
        A           - the first Structure , which may be the output of Structure , Union,
            Subtract , or Intersect
        B           - the second Structure , which may be the output of Structure , Union,
            Subtract , or Intersect
    @returns:
        structure   - the new structure object containing the intersection of A and B
    @example_usage:
        final_obj = Intersect (c_obj , s_obj )



==================================
    Prebuilt Convex Polytopes
==================================
There are 3 prebuilt convex polytopes (CP) available for use: cuboid, triPrism , and tet . Each CP
    comprises a set of Entities , namely faces , edges and corners .
For convenience, each individual entity can be referenced using the pattern <CP>.<entity_type
    >.<ENTITY_NAME>.
For example, you can select a particular edge of the cuboid with the notation cuboid.edges.
    BOTTOM_RIGHT.
Each CP also has an embed() method which returns all necessary information to embed the CP within
    R^3.

The full list of entities and embed() method signatures for our predefined CPs are as follows:

tet . corners .{    BOTTOM_RIGHT,
                    BOTTOM_LEFT,
                    TOP_BACK,
                    BOTTOM_BACK
                }
tet .edges. {       BOTTOM_FRONT,
                    TOP_LEFT,
                    BACK,
```

```
                    BOTTOM_RIGHT,
                    TOP_RIGHT,
                    BOTTOM_LEFT
                }
tet.faces. {    BOTTOM,
                    TOP,
                    RIGHT,
                    LEFT
                }
tet.embed(bounding_box_side_length)
    @description:
        Constructs the information required to embed the tet CP in R^3
    @params:
        bounding_box_side_length - length of axis-aligned bounding box containing the tet. Float in
            range [0,1]. Must be 1/2^k for some integer k
    @returns:
        embedding       - the embedding information. Specifically, the position in R^3 of all the
            CP corners.
    @example_usage:
        side_len = 0.5 / num_tiling_unit_repeats_per_dim
        embedding = tet.embed(side_len)


triPrism.corners.{FRONT_BOTTOM_LEFT,
                    FRONT_TOP,
                    FRONT_BOTTOM_RIGHT,
                    BACK_BOTTOM_LEFT,
                    BACK_TOP,
                    BACK_BOTTOM_RIGHT
                }
triPrism.edges.{FRONT_LEFT,
                    FRONT_RIGHT,
                    FRONT_BOTTOM,
                    BACK_LEFT,
                    BACK_RIGHT,
                    BACK_BOTTOM,
                    BOTTOM_LEFT,
                    TOP,
                    BOTTOM_RIGHT
                }
triPrism.faces.{FRONT_TRI,
                    BACK_TRI,
                    LEFT_QUAD,
                    RIGHT_QUAD,
                    BOTTOM_QUAD
                }
triPrism.embed(bounding_box_side_length)
    @description:
        Constructs the information required to embed the triangular prism CP in R^3
    @params:
        bounding_box_side_length - length of axis-aligned bounding box containing the triangular
            prism. Float in range [0,1]. Must be 1/2^k for some integer k
    @returns:
        embedding       - the embedding information. Specifically, the position in R^3 of all the
            CP corners.
    @example_usage:
        side_len = 0.5 / num_tiling_unit_repeats_per_dim
        embedding = triPrism.embed(side_len)


cuboid.corners.{FRONT_BOTTOM_LEFT,
                    FRONT_BOTTOM_RIGHT,
                    FRONT_TOP_LEFT,
                    FRONT_TOP_RIGHT,
                    BACK_BOTTOM_LEFT,
```

```
                    BACK_BOTTOM_RIGHT,
                    BACK_TOP_LEFT,
                    BACK_TOP_RIGHT
                }
cuboid.edges.{      FRONT_BOTTOM,
                    FRONT_LEFT,
                    FRONT_TOP,
                    FRONT_RIGHT,
                    BACK_BOTTOM,
                    BACK_LEFT,
                    BACK_TOP,
                    BACK_RIGHT,
                    BOTTOM_LEFT,
                    TOP_LEFT,
                    TOP_RIGHT,
                    BOTTOM_RIGHT
                }
cuboid.faces.{      FRONT,
                    BACK,
                    TOP,
                    BOTTOM,
                    LEFT,
                    RIGHT
                }

cuboid.embed(width, height, depth, cornerAtMinPt)
    @description:
        Constructs the information required to embed the cuboid CP in R^3
    @params:
        width           - length of cuboid side from left to right. float in range [0,1]. Must be
                1/2^k for some integer k
        height          - length of cuboid side from top to bottom. float in range [0,1]. Must be
                1/2^k for some integer k
        depth           - length of cuboid side from front to back. float in range [0,1]. Must be
                1/2^k for some integer k
        cornerAtMinPt   - CP corner entity (e.g., cuboid.corners.FRONT_BOTTOM_LEFT) that
                should be collocated with the cuboid's minimum position in R^3
    @returns:
        embedding       - the embedding information. Specifically, the position in R^3 of all the
                CP corners.
    @example_usage:
        side_len = 0.5 / num_tiling_unit_repeats_per_dim
        embedding = cuboid.embed(side_len, side_len, side_len, cornerAtAABBMin=cuboid.corners.
                FRONT_BOTTOM_LEFT)

cuboid.embed_via_minmax(aabb_min_pt, aabb_max_pt, cornerAtMinPt)
    @description:
        Constructs the information required to embed the cuboid CP in R^3
    @params:
        aabb_min_pt     - Minimum point of the cuboid, in R^3. Given as a list of length 3, where
                each component must be a float in range [0,1], with 1/2^k for some integer k
        aabb_max_pt     - Maximum point of the cuboid, in R^3. Given as a list of length 3, where
                each component must be a float in range [0,1], with 1/2^k for some integer k
        cornerAtMinPt   - CP corner entity (e.g., cuboid.corners.FRONT_BOTTOM_LEFT) that
                should be collocated with the cuboid's minimum position in R^3
    @returns:
        embedding       - the embedding information. Specifically, the position in R^3 of all the
                CP corners.
    @example_usage:
        side_len = 0.5 / num_tiling_unit_repeats_per_dim
        embedding = cuboid.embed([0,0,0], [side_len, side_len, side_len], cuboid.corners.
                BACK_BOTTOM_RIGHT)
```

API Errata The API description listed in this section is the exact version we used to train all models in MetaBench. This differs slightly from the released version, which corrects two mistakes that were identified at a later stage:

- `cuboid.embed()`: the original description (above) listed a parameter `cornerAtMinPt` in both the signature line and the `@params` listing. However, the `@example_usage` showed the parameter as `cornerAtAABBMin`. The latter is correct, and reflects an update made in the code independently of the documentation. The released API description consistently shows the correct parameter name, `cornerAtAABBMin`.
- `cuboid.embed_via_minmax()`: the `@example_usage` field of the original description (above) erroneously lists the `cuboid.embed()` function with the inputs of the intended function, `cuboid.embed_via_minmax()`. None of the parameters were updated, as they are all correct in the original description above. Only the erroneous function call was corrected in the released version (`cuboid.embed()` → `cuboid.embed_via_minmax()`).

These mistakes did not cause any observable issue in the trained model output, as the (correctly expressed) training data overrode the error in our API description. However, this did cause an issue for zero shot experiments (which ultimately revealed the bug). All zero shot results reported in the paper reflect the results using the *updated* version of our API, where the difference relative to the listing above constitutes exactly the two changes discussed here.

To ensure that this API description would not derail otherwise successful program outputs (and to mitigate confusion between the two very similar keywords across functions), we added an optional keyword argument to the signature of both affected functions, such that either keyword (or no keyword, as in a positional argument) is permissible. Thus, either API description is suitable; however, we release the corrected version to prevent issues and reduce confusion moving forward.

I.3 Reconstruction

Reconstruction tasks can have any combination of one to four views. Here we only reproduced the 4 view template; the others have the irrelevant lines removed.

```
# Task:
Analyze these views of a metamaterial, then generate a metamaterial DSL procedure to reproduce it.

# Inputs:
**Rendered Views:**
Top: <[{top}]>
Front: <[{front}]>
Right: <[{right}]>
Angled (Front-Top-Right): <[{top_right}]>

# Output Format:
Generate a Metagen program within a python code block:

```python
from metagen import *

def make_structure(...) -> Structure:
    ...
```
```

I.4 Inverse Design

```
# Task:
Write a metagen program that creates {query_target}.

# Output Format:
Generate a Metagen program within a python code block:
```

```
```python
from metagen import *

def make_structure(...) -> Structure:
    ...
```
```

I.5 MATERIAL UNDERSTANDING

Single View:

```
# Task:
Analyze these views of a metamaterial, and predict its material properties.

# Inputs:

**Rendered View:**

- Angled (Front-Top-Right): <[{top_right}]>

# Output Format:

Output a json object, delimited by ```json ```, where the keys are material property names, and the values are the predicted material properties. Predict these properties (keys):
- "A": Anisotropy (universal anisotropy index)
- "E": Young's Modulus relative to E_base
- "K": Bulk modulus relative to E_base
- "G": Shear modulus relative to E_base
- "nu": Isotropic Poisson ratio
- "V": Relative Density (Volume Fraction)
```

Multiview + Code:

```
# Task:
Analyze these views of a metamaterial, and the Metagen program, and predict its material properties.

# Inputs:

**Metagen Program:**

{code}

**Rendered Views:**
- Top: <[{top}]>
- Front: <[{front}]>
- Right: <[{right}]>
- Angled (Front-Top-Right): <[{top_right}]>

# Output Format:

Output a json object, delimited by ```json ```, where the keys are material property names, and the values are the predicted material properties. Predict these properties (keys):
- "A": Anisotropy (universal anisotropy index)
- "E": Young's Modulus relative to E_base
- "K": Bulk modulus relative to E_base
- "G": Shear modulus relative to E_base
- "nu": Isotropic Poisson ratio
- "V": Relative Density (Volume Fraction)
```

J Effect of Model Merging on Task Performance

To preserve prior capabilities after target-domain tuning (e.g., safety behaviors, general reasoning, coding, tool-use), a post-hoc *merging* step is generally employed that interpolates the target fine-tuned weights with the original base model. In the main results (Table 1), we report the merged variant. However, follow-up controlled experiments indicate that an *unmerged* model can exceed the merged model on target-domain benchmarks. This appendix section formalizes the construction and quantifies the effect.

Model	Material Understanding Error ↓	Reconstruction CD ↓	Reconstruction IoU ↑	Reconstruction Valid ↑
Model 1 (merge w/ base)	0.021 ± 0.003	0.035 ± 0.001	0.449 ± 0.007	**97.5%** $\pm$ 0.4%
Model 2 (no merge)	**0.015** ± 0.002	**0.028** ± 0.001	**0.533** ± 0.008	96.4% $\pm$ 0.4%
Δ (2 − 1)	-0.006	-0.007	$+0.084$	-1.1 pp

Table 9: Effect of model merging on task metrics in the Nova Model. Model 1 merges the target fine-tune with the base model; Model 2 omits merging. Best values per column are in **bold**. "pp" denotes percentage points.

Let θ_{base} denote the base parameters and θ_{tgt} the target-domain fine-tuned parameters. Model 1 is constructed by weight-space interpolation with the base,

$$\theta_{\text{merge}} \;=\; (1-\lambda)\,\theta_{\text{tgt}} + \lambda\,\theta_{\text{base}} \;=\; \theta_{\text{base}} + (1-\lambda)\big(\theta_{\text{tgt}} - \theta_{\text{base}}\big), \quad \lambda = 0.6,$$

which shrinks the task-specific delta toward the base. Model 2 uses $\lambda = 0$ (no merge). All data, compute, and optimization hyperparameters are held fixed; the only difference is the merge.

Table **9** [2] shows that removing the merge yields consistently better target-domain reconstruction: *CD* improves from 0.035 ± 0.001 to 0.028 ± 0.001 (relative $\downarrow 20\%$), and *IoU* rises from 0.449 ± 0.007 to 0.533 ± 0.008 (relative $+18.7\%$). Auxiliary tasks move in the same or neutral direction: *Material Understanding Error* decreases from 0.021 ± 0.003 to 0.015 ± 0.002 (Valid $= 100\%$ for both). Together, these effect sizes are large relative to the reported uncertainty and are aligned with the removal of the merge (λ: $> 0 \rightarrow 0$).

Interpolating toward θ_{base} creates a trade-off: while it can preserve broader domain capabilities, it dilutes the beneficial target-specific adaptations and reintroduces base model behaviors that perform poorly on the target task. A local quadratic approximation around θ_{tgt} with Hessian H yields $\mathcal{L}(\theta_{\text{merge}}) - \mathcal{L}(\theta_{\text{tgt}}) \approx \frac{1}{2}\lambda^2 \|\theta_{\text{tgt}} - \theta_{\text{base}}\|_H^2$, predicting systematic degradation as λ increases. The empirical ordering (Model 2 $>$ Model 1 on CD/IoU) and the magnitude of the gains are therefore most parsimoniously explained by the *merging step* rather than by data, compute, or randomness.

Under our target focused setting, merging the target fine-tune with the base model dilutes specialization and materially harms reconstruction (CD $\downarrow 20\%$, IoU $\uparrow 18.7\%$ when omitting the merge). Consequently, we recommend λ=0 for target-centric deployments; alternative merge recipes may aid robustness/multitask breadth, but they are unnecessary—and harmful—for this target-domain objective.

[2] Lower is better for Error and CD; higher is better for IoU and Valid.