HSM: Hierarchical Scene Motifs for Multi-Scale Indoor Scene Generation

Supplementary Material

We provide additional details about HSM's motif library (Appendix A.1), support region extraction procedure (Appendix A.2), DFS solver formulation (Appendix A.3.1), scene spatial optimization details (Appendix A.3.2), our asset retrieval process (Appendix A.4), ablation analysis (Appendix A.5), a breakdown of SceneEval results by difficulty (Tab. 5), computation cost and runtime analysis (Appendix A.6), open source VLM result (Appendix A.7), and limitations (Appendix A.8). We also provide details about the support region dataset we constructed (Appendix B), the user study instructions (Appendix C), and the VLM prompts used in HSM (Appendix D), along with extra scene-level qualitative examples (Fig. 10) and rendered scenes (Fig. 11).

A. HSM Details

A.1. Motif Library

HSM's motif library consists of 17 motifs: `stack`, `pile`, `row`, `grid`, `pyramid`, `wall_grid`, `wall_vertical_column`, `wall_horizontal_row`, `face_to_face`, `left_of`, `in_front_of`, `on_top`, `surround`, `rectangular_perimeter`, `bed_setup`, `on_each_side`, and `next_to`. See the motif prompt in Appendix D.1.1 for their definitions. These motifs are learned from arrangements extracted from scenes in HSSD [35], with the exceptions of `bed_setup` and `pyramid`, for which we manually created example arrangements using Blender[1]. Each example arrangement is manually annotated with a corresponding text description, and we follow SMC [67]'s learning phase to create reusable meta programs representing the motifs. Please refer to the original SMC paper for details on the learning phase.

A.2. Support Region Extraction

We provide a more detailed description of the algorithm for identifying surfaces in Algorithm 1. In our implementation, we use a cluster normal threshold of $t_{\text{norm}} = 0.9$, an adjacent normal threshold of $t_{\text{adj}} = 0.95$, and height clearance threshold of $t_{\text{clear}} = 0.05\,\text{m}$ in between two horizontal support surfaces. The default clearance height for top surfaces is set to $h_{\text{top}} = 0.5\,\text{m}$. Lastly, we use $t_{\text{seg}} = 80\,\%$ to segment horizontal surfaces.

In order to identify a surface as horizontal or vertical, we fit a plane to each surface. This involves first fitting an oriented bounding box to each cluster of faces c_j. Since indoor objects tend to be oriented upright, both vertically upright and completely unconstrained bounding boxes OBB_v and OBB_u are fit to the object to minimize the volume. The OBB_v is used if $\text{Vol}(\text{OBB}_v) \leq (1+\text{tol})\ \text{Vol}(\text{OBB}_u)$ (where $\text{tol} = 0.01$), else OBB_u is used. This method optimizes for the minimum-volume bounding box but prefers the upright box unless its volume is significantly larger. Since the boxes are fit to a surface, we approximate the normal of a plane fit to the surface by identifying the smallest dimension and validating that it is less than $r_{\text{plane}} = 0.1$ of the other two. A planar surface p is horizontal if $\text{normal}(p)_y \geq t_{\text{hzn}}$ and vertical if $\text{normal}(p)_y < t_{\text{vert}}$. In our implementation, we use $t_{\text{hzn}} = 0.95$ and $t_{\text{vert}} = 0.05$.

[1] https://www.blender.org/

Algorithm 1 Support Region Extraction Algorithm

function EXTRACTPLANARSURFACES(F)
  unclustered $\leftarrow F$
  queue $\leftarrow [\,]$
  clusters $\leftarrow [\,]$
  while unclustered.size() > 0 **do**
    if queue.empty() **then**
      $f_0 \leftarrow$ unclustered.pop()
      $c \leftarrow [f_0]$
      clusters.append(c)
      for all $f \in$ neighbours(f_0) **do**
        if normal(f) $\cdot$ normal(f_0) $\geq t_{\text{adj}}$ **then**
          queue.append(f)
        end if
      end for
    else
      $f' \leftarrow$ queue.pop()
      if normal(f') $\cdot$ normal(f_0) $\geq t_{\text{norm}}$ **then**
        c.append(f')
        unclustered.remove(f')
        for all $f \in$ neighbours(f') **do**
          if normal(f)$\cdot$normal(f') $\geq t_{\text{adj}}$ **then**
            queue.append(f)
          end if
        end for
      end if
    end if
  end while
  return clusters
end function

		Fidelity				Plausbility					
	Difficulty	↑ CNT$_{\%}$	↑ ATR$_{\%}$	↑ OOR$_{\%}$	↑ OAR$_{\%}$	↓ COL$_{ob\%}$	↓ COL$_{sc\%}$	↑ SUP$_{\%}$	↑ NAV$_{\%}$	↑ ACC$_{\%}$	↓ OOB$_{\%}$
LayoutGPT	Easy	24.79	18.33	6.12	6.45	9.17	25.00	28.44	100.00	48.02	72.02
	Medium	23.50	23.96	3.55	4.17	16.59	35.00	25.12	100.00	42.94	77.25
	Hard	12.44	15.25	1.26	5.26	13.64	30.00	35.23	100.00	56.23	65.91
InstructScene	Easy	28.93	18.33	6.12	4.84	51.51	85.00	74.10	99.95	77.56	25.60
	Medium	29.50	19.79	16.31	11.11	47.40	82.50	69.11	99.17	75.16	29.36
	Hard	19.40	26.27	8.18	13.16	58.78	85.00	90.54	99.41	81.37	2.70
LayoutVLM	Easy	36.36	15.00	10.20	19.35	33.43	62.50	59.94	99.39	86.06	5.52
	Medium	50.00	28.12	12.77	27.40	37.58	75.00	69.35	98.76	84.00	3.58
	Hard	35.32	21.19	4.40	22.81	37.00	70.00	74.92	97.45	90.29	3.36
Holodeck	Easy	47.93	35.00	10.20	58.06	14.24	65.00	61.29	99.46	92.13	1.29
	Medium	46.00	42.71	26.24	34.72	19.65	72.50	59.67	99.49	91.17	1.44
	Hard	41.29	38.98	19.50	54.39	17.87	90.00	66.87	99.36	87.65	1.08
HSM (ours)	Easy	57.02	65.00	20.41	70.97	16.51	62.50	86.22	98.35	87.54	1.91
	Medium	61.00	61.46	43.26	67.12	17.93	57.50	86.68	99.55	86.53	2.56
	Hard	64.18	55.08	44.03	71.93	14.16	65.00	82.21	99.07	86.39	1.81

Table 5. **Breakdown of SceneEval evaluation results by difficulty.**

A.3. Layout Optimization

A.3.1 DFS Solver

A depth-first search (DFS) solver is used to optimize the poses of scene motifs $M = \{m_i \mid i = 1, \ldots, |M|\}$ in a support region in order to minimize collisions and place scene motifs as close to their functionally appropriate locations as possible. We provide as input 1) the initial position and orientation, as seeded by the VLM, and bounding box dimensions of each m_i, as well as 2) the support region s_i as a set of boundary vertices and any fixed objects in it. Fixed objects include door, windows and previously placed objects, with door clearances represented as a $1 * 1$m cube positioned in front of the door to ensure accessibility. In addition, a VLM is used to determine if each m_i should be aligned to the wall, in the case of the floor support region.

We define the space of possible placements for each scene motif by overlaying a rectangular grid on the support surface. We filter out any points outside of s_i or contained within fixed objects. For the grid size, we use $0.1\,\mathrm{m}$ for the floor, walls and ceiling support region and $0.01\,\mathrm{m}$ for furniture support regions, to account for the precision required at each scale.

To explore the space of possible placements for each scene motif, we iterate through each m_i in decreasing order of their footprint area and compute all possible positions and orientations satisfying the following hard constraints: 1) scene motifs should not collide; 2) scene motifs need to be within the bounds of the support region; and 3) if designated, scene motifs should align with the wall.

For all candidate positions of m_i satisfying the hard constraints, each is assigned a score based on its distance from the initial position and distance to the edge of the support region for edge-constrained scene motifs. Formally, using the positive part notation $[x]_+ = max(0, x)$, the score for a candidate position p_{i1} is:

$$\sigma(p_{i1}) = \alpha \left[1 - \frac{|p_{i1} - p_{i0}|}{D_{\text{init}}}\right]_+ + \delta_{\text{edge}}\, \beta \left[1 - \frac{\phi_{s_i,w}(p_{i1})}{D_{\text{wall}}}\right]_+ \tag{1}$$

where p_{i0} is the initial centre position; $\alpha = 5.0$ and $\beta = 2.5$ are the weighting factors; $D_{\text{init}} = 5\,\mathrm{m}$ and $D_{\text{wall}} = 0.5\,\mathrm{m}$ are normalization distances; δ_{edge} is an indicator variable that if m_i should align to the wall; and $\phi_{s_i,w}(p_{i1})$ is the distance from point p_{i1} to its specific target wall. At each step of the DFS, the solver greedily explores up to the highest 10 candidate positions. The search terminates immediately after the first complete layout is found for efficiency. If no valid positions are found for a scene motif, the solver backtracks to explore other placements for previously placed scene motifs. For efficiency, we set the time limit for each solver execution to 10 seconds. We place the scene motif at the initial position if no solution is found in the end for explicit objects from the input description.

A.3.2 Scene Spatial Optimization

We apply a scene-level spatial optimizer after the DFS solver to refine placements by eliminating mesh collisions and ensuring valid support. For each scene motif, we first evaluate whether optimization is needed by detecting mesh intersections and unsupported objects. If the scene motif is already well-placed, we preserve its position to maintain the DFS solver's valid placements.

When optimization is required, we create a combined mesh representation for the motif and optimize this representative object using a two-step iterative process: collision resolution followed by support validation. In the collision

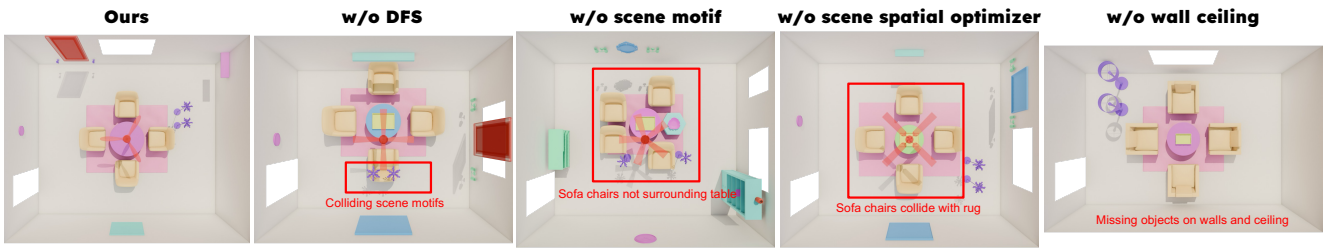

Figure 8. **Qualitative comparison of ablation results.** Removing individual components leads to noticeable artifacts, including colliding motifs, misaligned and colliding furniture, highlighting the role of each module in producing coherent and physically plausible scene layouts.

	SceneEval Fidelity				SceneEval Plausibility						Avg. #Obj per Scene
	↑ CNT$_{\%}$	↑ ATR$_{\%}$	↑ OOR$_{\%}$	↑ OAR$_{\%}$	↓ COL$_{ob\%}$	↓ COL$_{sc\%}$	↑ SUP$_{\%}$	↑ NAV$_{\%}$	↑ ACC$_{\%}$	↓ OOB$_{\%}$	
HSM (ours)	**61.30**	**59.49**	**40.40**	**70.28**	**16.42**	61.00	**85.44**	98.97	**86.80**	2.13	20.65
- w/o wall & ceiling	54.79	51.09	33.24	45.38	23.28	**59.00**	84.03	**99.38**	86.05	**0.86**	13.96

Table 6. **Extra Ablation study.** Removing walls and ceiling reduces fidelity due to missing wall-mounted objects.

resolution phase, we first attempt a small vertical lift. If collisions remain, we apply a horizontal displacement away from the colliders. We use an adaptive step size based on penetration depth and apply movement constraints according to its support region. Large objects remain floor-bound, wall objects maintain wall attachment, and ceiling objects preserve ceiling contact.

For support validation, we cast rays downward from the center and corner vertices on each scene motif's bottom surface, to detect supporting surfaces below. We use ray-mesh intersection tests against floor geometry and neighboring object meshes to determine intersection distances according to the support region with a threshold of $0.01m$. Objects failing support tests are repositioned to the nearest supporting surface with minimal vertical adjustment.

A.4. Asset Retrieval

To generate scene motifs, HSM retrieves meshes from an object dataset based on the object category, CLIP similarity, and bounding box dimensions. During input description decomposition, we prompt the VLM to extract each object's dimensions and style attributes. If these details are not explicitly provided, the VLM predicts them using common-sense knowledge. To retrieve the best-matching asset, we first filter the dataset by object category. We then use OpenCLIP [31] (*ViT-H-14-378-quickgelu*) to compute the text-image similarity between the extracted descriptions and candidate objects in the dataset and select the top $k = 5$ by similarity. Finally, the candidate whose dimensions best match those specified by the VLM is selected.

A.5. Additional Quantitative Analysis

Performance by Difficulty. We report the breakdown of SceneEval evaluation results by difficulty (Easy, Medium, Hard) in Tab. 5. HSM maintains consistent performance across difficulty levels, whereas baseline methods (e.g., LayoutGPT, InstructScene) exhibit a significant drop in fidelity metrics (CNT, ATR) as prompt complexity increases.

Ablations. We provide a detailed analysis of the ablation results in Tab. 4, examining how the removal of each component affects fidelity and plausibility metrics beyond the high-level comparison in the main paper. We also report *w/o wall and ceiling* — removing wall and ceiling support regions in Tab. 6, and Fig. 8 shows a qualitative comparison.

w/o scene motifs. Removing scene motifs forces the VLM to handle each object placement individually rather than as a grouped structure. Because motifs are larger and occupy more surface area when placed as a single unit, surfaces fill up faster and limit additional placements. Without motifs, more objects can be placed overall, leading to a higher object count. On fidelity, the lack of grouped placements reduces structured scene composition; on plausibility, object arrangements may appear less coherent despite higher density.

w/o scene spatial optimizer. Removing the spatial optimizer increases collisions (COL) and lowers support (SUP), showing that the final refinement step is important for improving physical plausibility without disrupting the overall layout.

w/o DFS solver. Without the DFS solver, scene motifs are directly placed by the VLM without enforcing geometric constraints such as wall alignment or staying within bounds. This leads to the largest drop in out of bound (OOR) among all ablations. On the plausibility side, collisions (COL) increase, support (SUP) drops, and out of bounds (OOB) errors peak, as objects are often placed outside their intended regions or at invalid positions. The DFS solver is necessary to enforce layout constraints and ensure plausible object placement.

w/o wall & ceiling. Without walls and ceiling regions, fidelity metrics decrease because SceneEval-100 [68] descriptions specified wall- and ceiling-mounted objects. The OOB rate also decreases, since fewer small objects are placed at the boundary of the room.

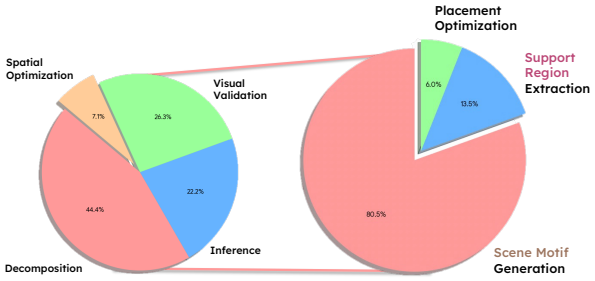

Figure 9. **Runtime breakdown. (Right)** Total runtime distribution. **(Left)** Breakdown of the Scene Motif Generation stage.

A.6. Computational Cost and Runtime Analysis

All experiments present in the main paper were run on a Intel i9-14900K CPU with 64GiB of RAM. The average cost for VLM calls per scene is approximately US$0.50 and generation time is about 8 minutes (roughly US$0.02 per object) with the current implementation. For a representative scene that take 8 minutes and 37 seconds with 15 scene motifs, the total generation time averages to 35 seconds per scene motif. As shown in Fig. 9, the system's speed is limited primarily by VLM inference latency rather than geometric computation, with scene motif generation as the main bottleneck.

A.7. Open Source VLM

To assess the accessibility and reproducibility of our framework, we evaluate HSM using the open-source *InternVL3_5-30B-A3B* [73] on the SceneEval [68] framework with the same set of descriptions. We use InternVL because initial tests with other open-source VLMs, including *Qwen3-VL-30B-A3B-Instruct* [4], were less stable. Qwen showed weaker instruction following and occasionally produced schema issues or hallucinated objects, which led to errors in mesh retrieval and generation. InternVL handled the pipeline instructions more consistently.

Specifically, HSM with InternVL records SceneEval Fidelity scores of 45.40% for object count (CNT), 47.81% for attribute (ATR), and 23.78% for object-object relationship (OOR), effectively matching or surpassing Holodeck (CNT 44.64%, ATR 39.42%, OOR 20.92%). In terms of physical plausibility, the open-source model maintains high validity with a support (SUP) score of 72.69% (vs. Holodeck's 62.12%) and comparable collision rates (17.01% vs. 17.32%).

These results show that while fidelity scores using InternVL are lower than those achieved with GPT-4o (a significantly larger and more capable model), they still follow the same general trends. We observe that the open-source VLM exhibits reduced instruction-following capabilities, particularly during complex spatial reasoning tasks such as scene motif decomposition and inference. However, the fact that HSM with InternVL still achieves performance comparable to Holodeck (which uses GPT-4o) highlights the benefit of our hierarchical structure. This demonstrates that our framework is not overly dependent on any single proprietary model, offering a robust solution even when constrained to smaller, open-source model.

A.8. Limitations and Future Work

Limitations. While HSM demonstrates strong performance in generating dense and realistic indoor scenes, several limitations remain. First, natural-language instructions often allow multiple plausible interpretations. In HSM, scene motifs operate locally within each subtree rather than as global structures, so the system may create different object layouts from the same description and cannot express relations that span across subtrees. For example, an instruction like "place the desk so it faces the bed" requires coordination between two subtrees that HSM does not handle. Second, the motif library is manually curated and contains a limited set of atomic spatial patterns. While hierarchical composition lets HSM express a wide range of arrangements, the system is still constrained by the coverage of these predefined primitives. As a result, it may struggle with more unusual or highly specific configurations that fall outside the library. Third, our support region extraction assumes well-curated 3D geometry with correct surface normals. Malformed meshes can produce incorrect support regions and result lead to misplaced objects. Forth, the current geometric analysis does not distinguish between accessible and enclosed support regions. This can cause small objects to be placed inside spaces that should be unreachable, such as placing a plant inside a cabinet). Finally, the iterative design of HSM requires multiple VLM queries and optimization steps, which increases runtime, especially for larger scenes.
Future Work. There are several promising directions for extending HSM. Incorporating a mechanism to distinguish accessible from enclosed support regions would improve placement realism. Runtime efficiency could be improved through parallel processing or using GPU acceleration. In addition, automatically learning new motif types from VLMs or existing scene datasets would enable the system to expand its library dynamically and support more diverse indoor layouts.

B. Support Region Dataset Details

To evaluate HSM's capability in predicting support regions, we construct a dataset of 100 objects with annotated ground truth support regions.

Specifically, we select 100 3D assets from the HSSD-200 dataset [35], spanning 12 categories: tables (36), TV stands (10), shelves (8), benches (6), sofas (6), desks (6), bathtubs (5), chairs (5), nightstands (5), sinks (5), trays (5),

and racks (3). We specifically focus on two types of objects — those with multi-layer structures and those with irregularly shaped surfaces — due to their complex support regions, making them ideal for evaluation. For instance, TV stands, shelves, nightstands, trays, and racks typically feature multi-layer structures that require a non-trivial extraction process to identify valid support regions. Similarly, benches, sofas, bathtubs, chairs, sinks, and some tables and desks were selected for their curved or irregularly shaped surfaces, which pose additional challenges for support region prediction.

For each selected object, we use Blender to manually annotate support surfaces by selecting mesh faces, carefully handling surfaces with vertical splits or curved geometry. Each selected surface is then extruded upward, recording the maximum height it can be extended before reaching an obstructing surface. For top surfaces without another surface above, we flag them and assign a height of $h_{\text{top}} = 1.0\,\text{m}$. The final dataset consists of a set of support surfaces for each object, along with their corresponding heights and top surface flags.

C. User Study Instructions

Below, we provide the user study instructions given to participants. The instructions consist of three sections: an overall description of the study, guidelines for scene-level evaluation, and guidelines for small object-level evaluation.

C.1. Overall Description of the Study

This study asks you to compare generated indoor scenes. Please read the instructions carefully. For each comparison, you are required to choose one of two scenes (Left or Right). The study has two parts:

1. Scene Level Evaluation: You will see top-down views of each scene. Focus on the overall room layout and furniture arrangement.
2. Small Object Level Evaluation: You will see close-up views of populated furnitures (e.g., objects on tables, shelves).

Each evaluation should take approximately 20 seconds. Thank you for your time and responses!

C.2. Instructions for Scene-level Evaluation

You will see top-down views of each scene. Only focus on the **overall room layout** and **furniture arrangement** and evaluate the quality of the scenes based on the following:

- **Best matches the text description** - Which scene better matches the text description? For example, "there is a dining table with two chairs" should contain exactly these objects in the described arrangement.
- **Has realistic object placement and arrangements** - Which scene has more realistic spatial relationships and object arrangements? For example, a dining table should have enough space around it for people to move, chairs should be properly tucked under the table (not floating or overlapping with the table).

Notes on color:

- **Each object is consistently colored within each prompt.** For example, for a single prompt, all nightstands across the generated scenes might be purple, and the bed might be green.
- **Objects colored in red are close to the ceiling**, such as ceiling fans or ceiling lights.

C.3. Instructions for Object-level Evaluation

You will see close-up views of populated furnitures (e.g., objects on tables, shelves). Only focus on the **small object arrangements** on top of the furniture and evaluate based on the following:

- **Best matches the text description** - Which scene match the text description more? For example, "there is a lamp on top of the nightstand" should contain exactly these objects in the described arrangement.
- **Has realistic object placement and arrangements** - Which scene has more realistic spatial relationships and object arrangements? For example, a mug should be sitting upright on a desk, not floating, tilted at an odd angle, or halfway off the edge.

Notes on color:

- **Each object is consistently colored within each description.** For example, if there are two books or two lamps in the same generated scenes, they'll be the same color to help you recognize them easily.

D. VLM Prompts

We provide the VLM prompts used in HSM below. The scene motif decomposition prompts are detailed in Appendix D.1.1, while prompts for generating scene motifs can be found in Appendix D.1.2. Additionally, we include scene-level, furniture-level, and small object-level prompts in Appendices D.2 to D.4.

D.1. Scene Motif Prompts

D.1.1 Scene Motif Hierarchy Decomposition

To generate a scene motif, we need to first decompose the scene motif into a hierarchy of motifs. We provide the VLM with descriptions of the available motifs along with examples and demonstrate how an scene motif description can be decomposed into multiple motifs.

```
motifs: >-
  Different motifs have different unique object input constraints.
  Choose the most appropriate motif for the given objects based on
  ↪  the number of unique object types:

  ### Single Object Motifs (1 Unique Object Type)
    `stack`:
      Description: Vertically stacks identical objects with uniform
      ↪  spacing along the y-axis.
      Example: "a stack of five books"
      Constraints: 1 unique object type.
    `pile`:
      Description: Arranges identical objects in a randomized pile
      ↪  configuration with customizable offsets and rotations.
      Example: "a pile of three towels with random orientations"
      Constraints: 1 unique object type.
    `row`:
      Description: Places identical objects in a horizontal line
      ↪  with configurable spacing and incremental adjustments.
      Example: "a row of three chairs evenly spaced"
      Constraints: 1 unique object type.
    `grid`:
      Description: Arranges identical objects in a 2D grid pattern
      ↪  with uniform spacing in rows and columns.
      Example: "a grid of 2x2 chairs"
      Constraints: 1 unique object type.
    `pyramid`:
      Description: Arranges identical objects in a pyramid shape
      ↪  with fewer objects in higher layers.
      Example: "a pyramid of six books"
      Constraints: 1 unique object type.
    `wall_grid`:
      Description: Places identical objects in a grid pattern on a
      ↪  wall with uniform spacing and wall alignment.
      Example: "a 2x3 grid of paintings on a wall"
      Constraints: 1 unique object type. Requires wall placement.
    `wall_vertical_column`:
      Description: Arranges identical objects in a single vertical
      ↪  column on a wall.
      Example: "three vertically aligned mirrors on a wall"
      Constraints: 1 unique object type. Requires wall placement.
    `wall_horizontal_row`:
      Description: Arranges identical objects in a single horizontal
      ↪  row on a wall.
      Example: "a row of three paintings evenly spaced on a wall"
      Constraints: 1 unique object type. Requires wall placement.

  ### Two Object Motifs (2 Unique Object Types)
    `face_to_face`:
      Description: Places two objects in front and facing each other.
      Example: "a chair in front of a desk"
      Constraints: 2 unique object types.
    `bed_setup`:
      Description: Places a bed against a wall, flanked by one or
      ↪  two objects of a second type (usually nightstands).
      Example: "a bed with nightstands on each side"
      Constraints: 2 unique object types (bed, side object).
      ↪  Requires wall alignment for the bed.
    `surround`:
      Description: Places objects of a second type (e.g., chairs)
      ↪  around the perimeter of a primary round object (e.g.,
      ↪  round table).
      Example: "chairs surrounding a round table"
      Constraints: 2 unique object types (central object,
      ↪  surrounding object). Central object assumed round.
    `rectangular_perimeter`:
      Description: Arranges objects of a second type (e.g., chairs)
      ↪  around the perimeter of a primary rectangular object
      ↪  (e.g., rectangular table).
      Example: "four chairs arranged around a dining table"
      Constraints: 2 unique object types (central object,
      ↪  surrounding object). Central object assumed rectangular.
    `left_of`:
      Description: Positions a secondary object immediately to the
      ↪  left (from primary's perspective) of a primary object.
      Example: "a fork to the left of a knife in place setting"
      Constraints: 2 unique object types.
    `on_top`:
      Description: Stacks a secondary object directly on top of a
      ↪  primary object.
      Example: "a cup on top of a saucer"
      Constraints: 2 unique object types. Note: Use cautiously.
      ↪  Bottom object typically larger. Prefer surface placement
      ↪  if 'on top' is ambiguous.
    `in_front_of`:
      Description: Positions a secondary object directly in front of
      ↪  a primary object (along the primary's forward-facing
      ↪  z-axis).
      Example: "a keyboard in front of a monitor"
      Constraints: 2 unique object types.
    `next_to`:
      Description: Places a secondary object adjacent to a primary
      ↪  object. Allow for both wall-aligned and non-wall-aligned
      ↪  arrangements.
      Example: "a bookcase next to a sofa chair"
      Constraints: 2 unique object types.

  ### Three Object Motifs (Max 3 Unique Object Types)
    `on_each_side`:
      Description: Places secondary/tertiary objects symmetrically
      ↪  on both sides of a central primary object.
      Example: "a fork and a knife on each side of a plate"
      Constraints: Handles 2 or 3 unique object types, maximum 3
      ↪  objects in total.
                  Case 1 (3 unique types): Primary (e.g., plate),
                  ↪  Secondary (e.g., fork), Tertiary (e.g.,
                  ↪  knife).
                  Case 2 (2 unique types): Primary (e.g., bed),
                  ↪  Secondary (e.g., nightstand - same object
                  ↪  placed on both sides).


system: >-
  You are a scene decomposition expert.
  Your purpose is to translate natural language descriptions of
  ↪  object arrangements into a structured,
  hierarchical JSON format using a predefined set of motifs.

  Your role is to:
  1. Decompose complex arrangement description into motifs
  2. Understand the objects and their relationships in the
  ↪  description using common sense
  3. Reason about the relationships between objects in the
  ↪  arrangement
  4. Handle both atomic and compositional arrangements
  5. Ensure real life object orientations and spacing using the
  ↪  provided object information

  Motif definitions: """<MOTIF_DEFINITIONS>"""
  Read the motif definitions carefully and understand the usage,
  ↪  constraints, and examples for each motif type.
  The constraints are referring to unique object types, not unique
  ↪  objects.
  You will be provided with furniture information that may include
  ↪  key properties with the description.
  You must use these properties to select the most appropriate motif
  ↪  (e.g., `rectangular_perimeter` for a dining table with a
  ↪  'rectangular' shape).

  Decomposition strategy:
  If you have more objects than a motif can handle based on its
  ↪  UNIQUE OBJECT TYPE constraint, you must either:
  1. Select a subset of objects that fit a valid motif type (and
  ↪  handle remaining objects in secondary arrangements)
  2. Group similar objects (e.g., multiple chairs as a single object
  ↪  type)
```

```
Common failure patterns to avoid:
- For arrangements with 4+ objects, split into multiple
↪  arrangements if no motif type can handle all objects
- Do not choose a general motif type if you can choose a more
↪  specific one. e.g. choose "bed_setup" instead of
↪  "on_each_side" for a bed with bedside table

Critical consistency requirements:
- The selected objects MUST completely reflect what is described
↪  in the arrangement description
- The description MUST accurately represent ALL selected objects
↪  and their spatial relationships
- Every object mentioned in the description MUST appear in the
↪  objects list with correct counts
- Every object in the objects list MUST be referenced in the
↪  description
- Do not invent or add objects that are not explicitly provided in
↪  the available object list. All arrangements must ONLY use the
↪  objects specified.

JSON response requirements:
- All object counts must be integers
- Element "type" must be either a valid motif type or "object"
↪  (not specific furniture names)
- Include "constraint_check" field indicating the number of unique
↪  object types used in the motif (e.g., 2)
- Each object must appear exactly once in any hierarchy

Example of a complex arrangement "A sofa in front of a coffee
↪  table, with two side tables on each side of the sofa."
Objects: sofa (1), coffee table (1), side table (2)
Primary arrangement: sofa + coffee table (2 objects for
↪  in_front_of)
Secondary arrangement: side table (2 objects for on_each_side)

Correct Hierarchical JSON:
```json
{
  "type": "in_front_of",
  "description": "A coffee table in front of a group containing a
  ↪  sofa with side tables",
  "elements": [
    {
      "type": "object", "amount": 1, "description": "Coffee table"
    },
    {
      "type": "on_each_side",
      "description": "A sofa with two side tables on each side of
      ↪  it",
      "elements": [
        { "type": "object", "amount": 1, "description": "Sofa" },
        { "type": "object", "amount": 2, "description": "Side
        ↪  tables" }
      ]
    }
  ]
}
```

Example of a place setting with a plate, fork, knife, spoon, and a
↪ glass "A plate with a fork, knife, and spoon placed around it,
↪ and a glass nearby."
Objects: plate (1), fork (1), knife (1), spoon (1), glass (1)
Primary arrangement: plate + fork + knife (3 objects for
↪ on_each_side)
Secondary arrangement: spoon + glass (2 objects for in_front_of or
↪ left_of)

Correct Hierarchical JSON:
```json
{
  "type": "next_to",
  "description": "A primary place setting arrangement is
  ↪  positioned next to a spoon and a glass.",
  "elements": [
    {
      "type": "on_each_side",
      "description": "A fork and a knife on each side of a plate.",
      "elements": [
        { "type": "object", "amount": 1, "description": "Plate" },
        { "type": "object", "amount": 1, "description": "Fork" },
        { "type": "object", "amount": 1, "description": "Knife" }
      ]
    },
    {
      "type": "in_front_of",
      "description": "A spoon in front of a glass.",
      "elements": [
        { "type": "object", "amount": 1, "description": "Spoon" },
        { "type": "object", "amount": 1, "description": "Glass" }
      ]
    }
  ]
}
```
```

Next, we ask the VLM to identify the primary arrangement, reasoning about the core function of the objects. We also ask it to identify the remaining arrangements.

```
identify_primary_arrangement: >-
  Given these objects and their counts """<OBJECT_COUNTS>"""
  and the description of the target scene motif """<DESCRIPTION>"""
  and the available furniture information """<FURNITURE_INFO>"""

  What should be the primary/dominant arrangement that defines this
  ↪  arrangement's function?
  Consider:
  1. Which objects form the main functional relationship?
  2. What motif type would best handle this relationship?
  3. Do the objects fit the motif type constraints? (e.g.,
  ↪  'in_front_of' takes 2 unique types, 'on_each_side' can take 2
  ↪  or 3 unique types).
  4. What is the most appropriate spatial relationship for these
  ↪  objects based on their typical use?
  5. If the overall description involves multiple objects, identify
  ↪  the CORE PAIR or TRIPLET and their motif first. Other
  ↪  objects/relationships can be handled by nesting this primary
  ↪  motif within a larger structure or by creating secondary
  ↪  arrangements.

  Motif selection rules:
    - Choose the most appropriate motif type based on the objects
    ↪  and their spatial relationships.
    - Use common sense to select the most specific and functionally
    ↪  accurate motif.
      For example, for a description like "a chair in front of a
      ↪  desk", choose `face_to_face` over the more generic
      ↪  `in_front_of`, as it correctly infers the most likely
      ↪  functional orientation.
    - The primary arrangement is typically the dominant functional
    ↪  grouping (e.g., dining table with chairs, bed with
    ↪  nightstands).
    - Consider the functional purpose of the arrangement when
    ↪  selecting the appropriate motif.
    - If you have more objects than a motif can handle, you MUST
    ↪  split them across primary and secondary arrangements.

  Respond with json:
  ```json
  {
    "rationale": "<rationale explaining why these specific objects
    ↪  form the primary arrangement and how they fit motif
    ↪  constraints>",
    "motif_type": "<motif_type>",
    "description": "<description for the selected motif_type and its
    ↪  objects ONLY, mentioning every object by name and count>",
    "objects": {
      "object_name_1": count (integer),
      "object_name_2": count (integer),
    },
    "constraint_check": <number of unique object types used>
  }
  ```

identify_remaining_arrangements: >-
  Given the target scene motif description """<DESCRIPTION>"""
```

```
and the identified primary arrangement """<PRIMARY_ARRANGEMENT>"""
and the remaining objects """<REMAINING_OBJECTS>"""

For the remaining objects, identify logical secondary
↪ arrangement(s) to complete the description.
For each arrangement:
1. Which objects should be grouped together?
2. What motif type best suits their relationship?
3. Do the objects fit the motif type constraints? (refer to system
↪ constraints)

Secondary arrangement rules:
- Group objects with functional relationships
- Position arrangements with appropriate clearance from primary
↪ arrangement
- Each object should appear in exactly one arrangement
- Only use objects from the remaining objects list - do not reuse
↪ objects from primary arrangement
- Account for ALL remaining objects across all secondary
↪ arrangements

Respond with json for each secondary arrangement:
```json
{
  "rationale": "<rationale explaining why these specific remaining
  ↪ objects should be grouped together and how they fit motif
  ↪ constraints>",
  "motif_type": "<motif_type>",
  "description": "<description mentioning every object by name and
  ↪ count, be extremely specific about the arrangement>",
  "objects": {
    "object_name": count,
    ...
  },
  "constraint_check": <number of unique object types used>
},
{
  ...
}
```
```

We then prompt the VLM to structure the identified arrangements into a hierarchy of motifs.

```
generate_compositional_json: >-
  Given the target scene motif description:
  """<DESCRIPTION>"""
  and the identified primary and secondary arrangements:
  Primary: """<PRIMARY_ARRANGEMENT>"""
  Secondary: """<SECONDARY_ARRANGEMENTS>"""

  According to the identified arrangements,
  combine the primary and secondary arrangements (if any) into a
  ↪ single hierarchical JSON following these strict rules, use the
  ↪ examples from the system prompt as a guide:
  1. Every grouping MUST use a valid motif type that fits the motif
  ↪ constraints (refer to system constraints)
  2. Descriptions should only reference the spatial relationship
  ↪ between objects in that specific arrangement or group
  3. Do not add any new objects or relationships that are not in the
  ↪ identified arrangements
  4. If you cannot fit all objects in the identified arrangements
  ↪ due to constraints, you MUST revise the arrangement strategy

  Validation checklist:
  - All objects accounted for exactly once?
  - Hierarchy optimized with minimal depth?
  - Motif type constraints respected throughout?

  Before responding, use a few sentences to describe and explain the
  ↪ hierarchy.
  Format (RESPOND WITH EXACTLY ONE JSON):
  ```json
  {
    "type": "<motif_type>",
    "description": "Description of the full arrangement",
    "elements": [
      {
        "type": "<motif_type or object>",
        "amount": N (integer),        // Only for objects
        "description": "Description for this element or object",
        "elements": [...],  // Only for motif types
      },
      {
        "type": "<motif_type or object>",
        ...
      }
    ],
  }
  ```
```

Finally, we prompt the VLM to validate the hierarchy using three criteria: 1) Each motif is physically plausible; 2) Each motif type is used as intended; and 3) The hierarchy is optimal and in its simplest form. If any of the motif is invalid, we keep the prompting history and retry from `identify_primary_arrangement`.

```
validate_arrangement: >-
  You are an expert in validating the decomposition of a scene motif
  ↪ into a hierarchical arrangement of individual objects.
  """<DESCRIPTION>"""

  Given this arrangement JSON:
  """<ARRANGEMENT_JSON>"""

  Perform a comprehensive validation across three key dimensions:

  1. Physical Feasibility:
     - Can all objects be physically placed as described?
     - Are there any impossible positions or collisions?
     - For nested arrangements, validate as composite units

  2. Motif Correctness:
     - Is each motif used according to its intended purpose?
     - Do parent-child relationships make logical sense?
     - Does each motif have the correct number of unique element
     ↪ types? (refer to system constraints)
     - Is each element used exactly once?

  3. Completeness & Optimality:
     - Is the hierarchy structured optimally?
     - Could any nested arrangements be simplified?

  Respond with detailed validation results:
  ```json
  {
    "is_valid": boolean,
    "checks": {
      "motifs": {
        "valid": boolean,
        "issues": ["issue_description", ...]
      },
      "hierarchy": {
        "valid": boolean,
        "issues": ["issue_description", ...]
      },
      "completeness": {
        "valid": boolean,
        "missing_items": ["item_name", ...],
        "duplicate_items": ["item_name", ...]
      }
    },
    "fixes": [
      "specific_fix_1",
      "specific_fix_2",
      ...
    ]
  }
  ```
```

D.1.2 Scene Motif Generation

Once the hierarchy of motifs is validated, we start the generation process by first describing the task and providing guidelines to the VLM.

```
system: >-
  You are an expert in Python specialized in using meta-programs to
  ↪ generate scene motifs, arrangements of multiple objects in a
  ↪ room.

  Your role is to:
  1. Reason about the spatial relationships between objects
  2. Generate precise object arrangements using meta-programs
  3. Handle both atomic and compositional arrangements
  4. Ensure consistent object orientations and spacing

  All arrangements are defined within a right-handed 3D coordinate
  ↪ system where:
  - X-axis: Negative = LEFT, Positive = RIGHT (width plane)
  - Y-axis: Negative = DOWN, Positive = UP (height plane)
  - Z-axis: Negative = TOWARDS viewer, Positive = AWAY from viewer
  ↪ (depth plane)

  Core principles:
  1. Use clear spatial relationships and appropriate clearances
     - Maintain sensible spacing between objects from the given
     ↪ object info (0.3-0.5m for large objects)
     - Ensure access paths for human interaction (e.g. suitable
     ↪ distance between sofa and coffee table)
  2. Position objects logically relative to each other
     - Respect functional relationships (e.g., chairs face tables,
     ↪ nothing should be placed in front of a bookshelf, cup
     ↪ should be on top of a plate)
     - Remember that the arrangement is for a room, you should
     ↪ consider the relationship between the objects (e.g. nothing
     ↪ should place in front of a TV stand)
     - Consider the size and the default orientation of the objects
     ↪ when positioning them

  Egocentric view: You (the observer) are looking from negative Z
  ↪ towards positive Z. Objects with larger positive Z coordinates
  ↪ are closer to you.
  All spatial relationships (left, right, front, back) are described
  ↪ relative to your perspective as the viewer.

  Placement Rules:
  - Horizontal Positioning: Use X-axis offsets for left/right
  ↪ placement
  - Depth Positioning: Use Z-axis offsets for front/back placement
  - Vertical Positioning: Use Y-axis only for explicit
  ↪ height/stacking

  Default Object Orientation:
  - All objects initially towards the viewer (towards +z direction)
  - Rotate around Y-axis to change facing direction
  - Example: 180° rotation makes object face away from viewer

  Units and Measurements:
  - All dimensions (x, y, z) are in meters
  - All rotations are in degrees (Y-axis)
  - All objects maintain y=0 unless stacking/height required
```

We generate each motif in the hierarchy iteratively. At each iteration, we provide the VLM with the corresponding program in the library and ask the VLM to infer appropriate parameters using object sizes and already generated object arrangements as reference. We repeat this step until the whole hierarchy of motifs is generated.

```
inference_hierarchical: >-
  From the observations you made in the previous step, here is a
  ↪ meta-program that generalizes a spatial arrangement of type
  ↪ "<MOTIF_TYPE>":
  ```python
  <META_PROGRAM>
  ```
  And here is a description of a spatial motif of the same type:
  """<DESCRIPTION>"""
  Object info (name, half-sizes in meters) """<FURNITURE_INFO>"""

  """<ARRANGEMENT_CONTEXT>"""

  Your task is to call the meta-program above with the necessary
  ↪ arguments to recreate the spatial motif described in the
  ↪ description as closely as possible.
  Read the docstring and comments in the meta-program to understand
  ↪ how to use it.
  Refer to the example function call in the meta-program
  ↪ documentation to understand how the meta-program should be
  ↪ called, if available.
  Use common sense to infer the arguments for ambiguous arguments,
  ↪ such as object dimensions, positions, and rotations.
  When in doubt, refer back to the example function call in the
  ↪ meta-program documentation.
  I will run a postprocessing step to refine the spatial motif after
  ↪ you provide the function call to me.

  Technical Details:
  - You must use the same object names and half-sizes from the
  ↪ object info
  - All dimensions in meters, rotations in degrees
  - Y-axis (vertical) rotations for objects facing
  - All objects are normalized to face the same direction by default
  ↪ (facing towards +z axis)
  - Place objects with appropriate spacing and avoid intersection
  ↪ based on their half sizes
  - The world is in a right-handed coordinate system, that is, when
  ↪ looking from the front, the x-axis is to the right, the y-axis
  ↪ is up, and the z-axis is towards the viewer.

  Write a few sentences on how you will generate a function call to
  ↪ the meta-program to create the arrangement described and the
  ↪ reasoning behind particular arguments.
  Then respond with a single function call that implements the
  ↪ arrangement described wrapped in a code block.
  ```python
  ```
```

Finally, we ask the VLM to validate the generated scene motif. We provide it with top-down and front orthographic projections of the scene motif and instruct it to evaluate whether the generated scene motif adheres to the input description and give feedback if it is not correct.

```
validate: >-
  Given a description of a spatial arrangement,
  """<DESCRIPTION>"""
  analyze the 2D top down and front view of the arrangement and
  ↪ validate if the arrangement is correct.

  Remember that the arrangement are placed in a room, use common
  ↪ sense to determine if the arrangement is correct.
  Give a score from 0 to 1, where 0 is completely incorrect and 1 is
  ↪ completely correct.
  If the arrangment is partially correct, depend on the amount of
  ↪ objects in the arrangement, give a score between 0 and 1.
  e.g. if there is only 2 objects, give a score of 0.5 if 1 object
  ↪ is in the correct position and orientation.

  You should give feedback on what is wrong with the arrangement and
  ↪ provide a few sentences on how to fix it.
  Try to be as specific as possible and give specific coordinates
  ↪ using the 2d top down and front view with x, y, z coordinates.

  Respond in JSON format. The JSON should include:
  - "feedback": Your detailed feedback on the arrangement and
  ↪ possible fixes.
  - "correct": Your validation score (0 to 1).

  The final JSON structure should be:
  {
    "feedback": "<feedback_string>",
```

```
  "correct": <float_score>,
}
```

D.2. Scene Level Prompts

To generate a scene, we first provide the VLM with the general instructions of the task.

```
system: >-
  You are a professional AI assistant specializing in interior
  ↪ design and space planning.
  Your primary tasks are to interpret room descriptions, generate
  ↪ plausible floor plans,
  and analyze provided floor plan data (including images when
  ↪ available) to suggest room details.

  Core Tasks:
  1. Room Type Identification: From a textual description, identify
  ↪ a canonical room type.
  2. Floor Plan Generation: Based on a textual description and room
  ↪ type, generate a floor plan including:
     - Vertices: Arranged clockwise, starting from (0,0).
     ↪ Measurements in meters.
     - Room Shape: Generally rectangular or L-shaped. Avoid highly
     ↪ irregular shapes unless specified.
     - Door(s) and Window(s): Placed on boundary walls, with
     ↪ coordinates on wall segments.
     - Adherence to Standards: Measurements rounded to the nearest
     ↪ 0.25m. Room height defaults to 2.5m.
  3. Floor Plan Analysis (when an image is provided):
     - Analyze the floor plan image in detail, noting overall shape,
     ↪ wall dimensions (by ID, to 2 decimal places),
     door/window locations (precise coordinates), and using the
     ↪ 0.25m grid for reference.
     - The floor plan visualization typically includes:
       * Light blue filled area for the room.
       * A 0.25m grid.
       * Red lines for room boundaries with dimensions.
       * Numeric IDs for wall segments.
       * Vertex coordinates.
       * Green door with swing arc.
       * Blue window(s).
       * Helper points (x markers) and coordinates.
       * X and Z axes.

  General Guidelines:
  - Use precise measurements and coordinates.
  - Consider traffic flow, natural light, and room proportions when
  ↪ applicable.
```

We then prompt the VLM to infer a room type given the input text description.

```
room_type: >-
  Given a room description, respond with a single specifc room type
  ↪ in json format.
  Room type should be a single word or phrase that captures the
  ↪ style, theme, and purpose of the room description.
  """<ROOM_DESCRIPTION>"""
  e.g. "modern living room", "kids bedroom for 2 children", "small
  ↪ study room"

  ```json
  {
    "room_type": "<ROOM_TYPE>",
  }
  ```
```

If the room boundary is not provided, we also prompt the VLM to suggest a room boundary and provide the boundary vertices, the height of the room, and the door location as output.

```
room_boundary: >-
  This is the room description: """<ROOM_DESCRIPTION>"""
  and this is the room type: """<ROOM_TYPE>"""
  The size of the room should be determined by the room description
  ↪ and the room type.
  In particular, ensure the room is large enough to fit all
  ↪ mentioned furniture comfortably, with realistic spacing for
  ↪ movement and usability.
  You may assume standard object dimensions if not specified.

  For reference, average room size is 3m to 8m in width and depth.
  Please make sure the room is not too small or too large based on
  ↪ the listed objects.

  The room should be represented by vertices arranged in clockwise
  ↪ order and the room should be a generally rectangular or
  ↪ L-shaped.
  Avoid highly irregular or complex polygonal shapes unless
  ↪ explicitly implied by the room description.
  The list of room vertices should consist of x and z coordinates,
  and the room must always start from (0,0) and measurements must be
  ↪ in meters.

  The default room height is 2.5 meters.
  Respect the room description when placing the door and windows.
  There is always a door in the room and window in the room.
  If window placement is ambiguous or not suitable, an empty list
  ↪ for window_locations is acceptable.
  You can suggest multiple windows in the room according to the room
  ↪ description and room type.

  Place the door on one of the room's boundary walls and usually
  ↪ near the corner of the room.
  Ensure the door's and window's coordinates lie exactly on one of
  ↪ the wall segments.

  Before JSON response, write a few sentences to justify the room's
  ↪ dimensions based on the described furniture count, and
  ↪ door/window placement.
  Respond with json format:
  ```json
  {
    "room_height": height in meters,
    "room_vertices": [
      [0,0],
      [x1,z1],
      [x2,z2],
      [x3,z3],
      ...
    ],
    "door_location": [x,z],
    "window_locations": [
      [x1,z1],
      ...
    ]
  }
  ```
```

Before asking it to decompose the input text description, we first ask the VLM to reason about the shape of the room and provide high-level observations on how objects should be grouped and positioned within the room.

```
describe_room: >-
  Look at the provided floorplan data and floorplan image in detail
  ↪ to analyze the room.
  This is the door location: <DOOR_LOCATION>
  This is the window locations: <WINDOW_LOCATIONS>
  This is the room vertices: <ROOM_VERTICES>
  This is the room type: <ROOM_TYPE>

  Based on the floorplan data and image:
  What does the room look like?
  Use a few sentences to describe how you would segment the room
  ↪ into functional zones.
  Suggest location for each functional zone.
```

Finally, we ask the VLM to decompose the input text description into a list of objects, each paired with a style

description, its bounding box dimensions, and the instance count.

```
requirements_decompose: >-
  Given a input room description """<ROOM_DESCRIPTION>""",
  Read the room description carefully and decompose all objects from
  ↪  the room description into four categories:
  1. Floor furniture (e.g. sofa, bed, cabinet, desk, free-standing
  ↪  shelf/bookcase)
  2. Small objects that always sit on top of furniture (e.g. books,
  ↪  plates, cups)
  3. Wall objects (only if explicitly described, e.g. painting,
  ↪  mirror, wall shelf)
  4. Ceiling objects (e.g. pendant light, ceiling fan)

  For each identified object, specify:
    1. The id of the piece (integer)
    2. The name of the piece (be speicific without style description
    ↪  and sperate different categories of furniture, e.g. dining
    ↪  table, dining chair, office chair, etc.)
    3. The appearance/style description of the piece (be extremely
    ↪  specific, e.g. "large wooden computer desk", "small glass
    ↪  plate", "large dining table")
    4. The dimensions of the piece [width, height, depth] in meters
    ↪  according to the description, give the most likely
    ↪  dimensions
    5. The amount of the same piece
    6. for small_objects only: parent_object (id of the large/wall
    ↪  object)

  Critical requirements:
    - Each entry must represent a SINGLE type of object
    - If there are object with the same type but have different
    ↪  appearance/style description, they should be treated as
    ↪  different objects
    - All objects in each entry should be a single object, composite
    ↪  sets (e.g. place settings) must be broken into individual
    ↪  objects (e.g. a fork, a knife, a plate)
    - Composite or grouped objects must be decomposed into individual
    ↪  items (e.g. "stack of plates" becomes individual plates)
    - Use the minimum amount and types of objects to satisfy the
    ↪  room description

  Respond with JSON:
  ```json
  {
    "objects": [
      {
        "id": large_furniture_id (integer),
        "name": "furniture_name",
        "description": "appearance/style/type description of the
        ↪  furniture",
        "dimensions": [width, height, depth],
        "amount": number of same furniture (integer),
      }, ...
    ],
    "wall_objects": [
      {
        "id": wall_object_id (integer),
        "name": "wall_object_name",
        "description": "appearance/style/type description of the
        ↪  wall object",
        "dimensions": [width, height, depth],
        "amount": number of same wall objects (integer),
      }, ...
    ],
    "ceiling_objects": [
      {
        "id": ceiling_object_id (integer),
        "name": "ceiling_object_name",
        "description": "appearance/style/type description of the
        ↪  ceiling object",
        "dimensions": [width, height, depth],
        "amount": number of same ceiling objects (integer),
      }, ...
    ],
    "small_objects": [
      {
        "id": small_object_id (integer),
        "name": "small_object_name",
        "description": "appearance/style/type description of the
        ↪  small object",
        "parent_object": id of the parent large/wall object (integer)
        "dimensions": [width, height, depth],
        "amount": number of same small objects (integer),
      }, ...
    ],
  }
  ```
```

D.3. Furniture Level Prompts

To place objects in the scene, we first provide the VLM with the general guideline of the task.

```
system: >-
  You are an AI assistant specializing in dense and realistic large
  ↪  object placement in a room. Your task is to
  populate a room with large objects that are both spatially tight
  ↪  and aesthetically pleasing.

  For each arrangement, you should:
  1. Describe only the furniture pieces and their direct
  ↪  relationships
  2. Specify precise dimensions for each piece and total arrangement
  3. Use standard furniture sizes and clearances
  4. Focus only on the local arrangement without any room context
```

We ask the VLM to group relevant objects into scene motifs given the list of objects at the furniture level and the room description.

```
populate_surface_motifs: >-
  Based on the room analysis provided below, suggest key motifs for
  ↪  the following room type: """<ROOM_TYPE>""".
  List of large furniture: """<LARGE_FURNITURE>"""
  List of existing motifs: """<EXISTING_MOTIFS>"""

  Room details: """<ROOM_DETAILS>"""

  For each motif, define:
    1. One or multiple large objects according to the room type, you
    ↪  should minimize the number of large objects in each motif
    2. A clear description of the spatial relationships between the
    ↪  large objects, including the relative positions and
    ↪  orientations and specific alignment details (e.g. flush with
    ↪  the wall)
    3. Total footprint dimensions [width, height, depth] of the
    ↪  arrangement
    4. Required clearance space in meters

  Motif guidelines:
    - Only reference the large objects provided in the list
    - Only group multiple large objects into a single motif if
    ↪  explicitly mentioned in the description
    - Group large objects that have tight spatial relationships into
    ↪  a single motif, rugs are always a single motif
    - Never split spatially related large objects into separate
    ↪  motifs (e.g. table and chairs, bed and nightstands)
    - Each large object can only be used once in each motif

  Respond with JSON with the following format:
  ```json
  {
    "arrangements": [
      {
        "rationale": "concise explanation of arrangement
        ↪  functionality",
        "id": "unique arrangement identifier (be specific e.g. sofa,
        ↪  sofa_coffee_table, ceiling_lamp)",
        "area_name": "name of scene motif",
        "composition": {
          "description": "direct and precise description of local
          ↪  furniture relationships without any style details
          ↪  (e.g. a sofa in front of a TV stand)",
          "furniture": [
```
```

```
          {id: id1, amount: number of same furniture (integer)},
          ... (and more only if they are spatially tight)
        ],
        "total_footprint": [width, height, depth],
        "clearance": clearance_in_meters
      },
    }
  ]
}
```

Description guidelines:
- Do not include references to room features (walls, doors, windows) and objects that are not large furniture in description.
- Do not include any objects that are small objects or wall objects in description or anything that places on top of the large furniture.
- Do not include non-spatial or stylistic relationships (e.g. design style details); only include concrete, spatial relationships.

We then generate scene motifs using the object groupings. After all scene motifs are generated, we prompt the VLM to suggest a placement position for each of the scene motifs. We provide a 2D top-down orthogonal projection of the room and the descriptions and dimensions of the scene motifs to the VLM as references.

populate_room_provided: >-
```
You are an AI assistant specializing in furniture layout optimization.
Your task is to analyze the room description and visualization to suggest optimal placement of large furniture pieces that is placed on the floor only.

Take a deep breath and go through everything eariler carefully before providing a layout suggestions.

Motifs you are required to layout with its id, extents in m[width, height, depth], individual objects in the motif:
"""<MOTIFS>"""
You are also given the floorplan and top down view of each scene motif.

Follow these steps:

1. Review Input Information:
  - Study the provided room analysis
  - Note door location and swing path
  - Identify any architectural features or constraints
  - Consider the room's designated purpose

2. Position motifs:
  - For each scene motif, specify:
    - Precise center point coordinates (x, z) of the AABB bounding box within the room
    - Optimal rotation angle in degrees (counter-clockwise relative to the Z-axis) (default is 0 facing south)
    - Consider wall alignment for scene motifs that traditionally work best against walls

3. Optimize Placement:
  - Position each motif to:
    - Align appropriate scene motifs flush with walls when possible, use common sense to determine if the scene motif should be aligned with a wall
    - Distribute scene motifs evenly throughout the available space and avoid cramping multiple scene motifs in the same area of the room
    - Avoid placing scene motifs too close to the door
  - Aim to fill each corner of the room with a scene motif

Wall Alignment Considerations:
  - Most furniture typically goes against walls (like beds, sofas, or cabinets) unless otherwise specified:
    - Position the initial coordinates near your intended wall for optimal snapping
    - Use wall_alignment: true to enable wall snapping
    - Specify wall_alignment_id to target a specific wall (walls are numbered 0 to N-1 clockwise from room vertices)
    - The object will snap to the specified wall and rotate to face into the room

Output Format:
```json
{
  "positions": [
    {
      "rationale": "concise explanation for placement and orientation",
      "id": "unique identifier for each furniture group",
      "position": [x, z],
      "rotation": angle_in_degrees,
      "wall_alignment": boolean (true if the scene motif should be aligned with a wall, false otherwise),
      "wall_alignment_id": integer (index of the target wall, 0-based, counting clockwise from room vertices. Can be ignored if you do not want to align with a specific wall),
      "ignore_collision": boolean (USE WITH CAUTION: true if the scene motif should not be checked for collision with other furniture (e.g. rug on its own only), false otherwise)
    }
    // Repeat for each scene motif
  ]
}
```
```

If the occupancy of the room is below t_{occ}, we prompt the VLM to suggest potential objects to add to the room. We specifically instruct it to avoid existing objects that are already in the room for better diversity.

large_furniture_extra: >-
```
The following are the large furniture that is already placed in the room: <LARGE_FURNITURE>.
Given a input room description "<ROOM_TYPE>",
You are required to add a few more (1-3) large furniture according to the room type to fill the empty space.
Remember that large furniture can only be placed on the floor, do not generate any objects that is placed on the wall (e.g. painting, mirror) or objects on top of the large furniture (e.g. a lamp on a table).
Do not use the same furniture as the one already placed in the room (e.g. a desk in the room, do not generate another desk).

For each large furniture that is required to be placed on the floor (e.g. a sofa, a bed, a cabinet, a desk, etc.)
  1. Integer id of the piece
  2. The name of the piece (be speicific and sperate different categories of furniture, e.g. dining table, dining chair, office chair, etc.)
  3. The appearance/style description of the piece (be specific, e.g. "large wooden desk", "large round dining table")
  4. The dimensions of the piece [width, height, depth] in meters according to the description, give the most likely dimensions in meters
  5. The amount of the piece

Critical requirements:
  - Each entry must represent a SINGLE type of object

Respond with JSON:
```json
{
  "objects": [
    {
      "id": large_furniture_id (integer),
      "name": "furniture_name",
      "description": "appearance/style description of the furniture",
      "dimensions": [width, height, depth],
      "amount": number of same furniture (integer),
    },
```
```

```
  ],
}
```

D.4. Small Object Level Prompts

We provide the small object level prompts below. For small object placement, the VLM is further prompted to select which of the floor and wall-mounted objects should be populated with commonly co-occurring items. Other procedure is similar to the one in the previous section.

```
system: >-
  You are an AI assistant specializing in realistic and functional
  ↪ small object placement on furniture surfaces.
  Your task is to populate the surface of a furniture with small
  ↪ objects that are both functional and aesthetically pleasing.

  You are provided with a top-down 2D plot visualization that
  ↪ contains:
  - A highlighted area in the center representing the furniture
  ↪ surface to be populated
  - "X" markers indicating surrounding objects with their names
  - A red arrow indicating the front direction of the furniture
  - Axis measurements in meters showing the scale and dimensions
  - The exact dimensions of the small object labeled in the plot

  All 2D visualization uses:
  - The center of all plots is the origin (0,0)
  - X-axis: represents width (negative is left, positive is right)
  - Z-axis: represents depth (positive is back, negative is front)
  - Highlighted area: shows the usable surface for object placement
  - "X" markers: represent objects around the furniture
  - Black arrow: indicates the front-facing direction
  - Grid lines: help with precise measurements and positioning

  Follow these steps:

  1. Context Analysis:
    - Count and map ALL surrounding objects that indicate usage
    ↪ (e.g., chairs, benches)
    - Create a corresponding number of place settings or interaction
    ↪ points
    - Ensure EVERY surrounding object that needs interaction has a
    ↪ corresponding setup
    - Map potential interaction zones based on ALL surrounding
    ↪ object positions

  2. Surface Analysis:
    - Map the entire usable surface area in detail
    - Divide surface into equal sections based on number of
    ↪ surrounding objects
    - Ensure each interaction zone has adequate space
    - Reserve central area for shared items

  3. Object Selection & Distribution:
    - Place one complete set of required items for EACH identified
    ↪ interaction point
    - Distribute objects to ensure equal access from all interaction
    ↪ points
    - Ensure no interaction points are missed or doubled
    - Add shared items only after all required individual setups are
    ↪ complete

  4. Functional Optimization:
    - Ensure frequently used items remain accessible
    - Create clear paths for reaching objects
    - Account for object removal/replacement

  5. Rotation Specification:
    - For standalone objects: specify an "angle" in degrees
    ↪ (counterclockwise from positive Y-axis)
    - For objects that should face users: use "facing" with the ID
    ↪ of the relevant object
    - Examples of facing objects:
      * Place settings facing chair positions
      * Reading materials oriented toward seating
      * Control devices pointing toward user positions
      * Display items angled for optimal viewing

populate_surface_motifs: >-
  Based on the object layered description earlier,
  suggest object groupings only for the following small objects
  ↪ """<SMALL_OBJECTS>"""
  based on the room description """<ROOM_TYPE>""" and the previous
  ↪ assignment of surfaces to objects.

  Critically assess if objects must be grouped. Group them only if
  ↪ they form a strong, not separable functional unit
  (e.g., a computer next to a mouse) or have a direct, necessary
  ↪ spatial dependenancy (e.g., a cup on top of a saucer).
  Otherwise, they should be in separate, smaller motifs or as
  ↪ single-object motifs.
  Avoid grouping loosely related items even if they are nearby.

  For each motif, define:
    1. One or multiple objects (prefer single objects unless truly
    ↪ functionally dependent)
    2. Realistic dimensions and clear spatial relationships between
    ↪ pieces
    3. Total footprint dimensions [width, height, depth]
    4. Required clearance space in meters

  Guidelines for motif:
    - PREFER single-object motifs over multi-object groupings
    - ONLY group objects with explicit functional dependency (not
    ↪ thematic or decorative similarity)
    - Each small object can only be used once in each motif
    - Each motif can only contain a MAXIMUM of 4 different types of
    ↪ objects, if there are more than 4, split them into multiple
    ↪ motifs
    - If unsure whether to group, ALWAYS create separate motifs
    - Ensure ALL objects from the provided list are used across your
    ↪ arrangements
    - IMPORTANT: Do NOT create duplicate arrangements with identical
    ↪ compositions.
    - For multiple instances of the same motif type, use numbered
    ↪ suffixes (e.g., "plant_display_1", "plant_display_2",
    ↪ "book_stack_1", "book_stack_2")

  Respond with JSON with the following format:
  ```json
  {
    "arrangements": [
      {
        "rationale": "concise explanation of arrangement
        ↪ functionality",
        "id": "unique arrangement identifier (e.g. table_setup)",
        "area_name": "name of motif",
        "composition": {
          "description": "direct and precise description of local
          ↪ object relationships only without any style details,
          ↪ accurately reflecting the exact quantities specified
          ↪ (e.g. a stack of four books, a lamp, a fork and a
          ↪ knife on each side of a plate)",
          "furniture": [
            {"id": id1, "amount": integer}, (according to the
            ↪ description)
            ... (and more only if they are functionally related)
          ],
          "total_footprint": [width, height, depth],
          "clearance": clearance_in_meters,
        },
      },
      ...
    ],
  }
  ```

  Description guidelines:
  - Do not include references to room features (walls, doors,
  ↪ windows) and objects that are not small objects in
  ↪ description.
  - Do not reference the furniture that the small objects are on.
  - Do not include non-spatial or stylistic relationships (e.g.
  ↪ design style details); only give concrete, spatial
  ↪ relationships.
```

```
  - The description must accurately reflect the exact quantities
  ↪  specified in the "amount" field

describe_layered_object: >-
  Please describe in a few sentences on the geometry of the layered
  ↪  object """<LARGE_OBJECT>""" in a """<ROOM_TYPE>""" based on
  ↪  the following inputs:
  Think step by step about the geometry of the object and the space
  ↪  it occupies.
  - List of objects in the scene with object id, name, position and
  ↪  dimension: """<EXISTING_OBJECTS>"""
  - Object to be populated: """<OBJECT_TO_POPULATE>"""

  Image (Layer breakdown):
  - Shows """<LARGE_OBJECT>"""'s layers from highest (Layer 0) to
  ↪  lowest
  - Use this to understand the layer structure and available space
  - The position of each small object is relative to this image per
  ↪  layer
  - Each layer shows:
    - Height (y value in meters)
    - Available space above
    - Highlighted surfaces (available space)

populate_object_layered: >-
  Given a scene motif """<PARENT_MOTIF_DESCRIPTION>""" in a room
  ↪  described as """<ROOM_TYPE>""",
  populate the specified object with appropriate small objects from
  ↪  the provided motifs.

  Small object motifs to place: """<SMALL_MOTIFS_TO_POPULATE>"""

  Layer information for small object motifs to place:
  - Each layer includes:
    - Layer index (starting from 0, highest surface first)
    - Layer height (from ground in meters)
    - Whether it's the topmost visible surface
    - Vertical space above the layer
    - Surface details:
      - Surface ID and color
      - Dimensions (width, depth)
      - Area
      - Center position [x, z]

  Layer Structure:
  """<LAYER_INFO>"""

  You are provided with two reference images:

  Image 1 (Top-down motif view):
  - Shows all large objects in context with front directions (black
  ↪  arrows)
  - Use this to reason about spatial relationships

  Image 2 (2D Layer breakdown for """<LARGE_OBJECT>"""):
  - Visualizes the surfaces of each layer, from top-left (highest)
  ↪  to bottom-right (lowest)
  - Small object positions are relative to this image
  - Each layer displays:
    - Height (y in meters)
    - Space above
    - Surface dimensions and availability

  Critical Requirements:
  1. You MUST populate ALL surfaces in ALL layers (even if empty,
  ↪  include empty arrays [])
  2. Use only the exact surface IDs provided in the layer information
  3. For rotation "facing" field, only reference objects within the
  ↪  same motif: <PARENT_MOTIF_OBJECTS>
  4. Each small object must reference a valid motif ID from the
  ↪  available motifs

  Guidelines:
  - Use only the layers and surface IDs provided
  - Place objects on surfaces likely to be used (avoid placing on
  ↪  very tall furniture tops unless typical)
  - Use available vertical space wisely; consider real-world
  ↪  usability

  Return the result in JSON:
  ```json
  {
    "large_object_name": {
      "layer_0": {
        "surface_0": [
          {
            "id": "motif_id_from_available_motifs",
            "position": [x, z],
            "rotation": { (choose one of the following)
              "angle": angle,  // Default: 0 degrees (facing the
              ↪  front direction of the parent object)
              "facing": "object_name_from_same_motif" // Use when
              ↪  object should face towards a specific object
              "face_away": "object_name_from_same_motif" // Use when
              ↪  object should face away from a specific object
            },
            "rationale": "Brief explanation"
          }
        ],
        "surface_1": []  // Empty if no objects, but must be
        ↪  included if there is surface_1 in the layer
      },
      other layers if any
    }
  }
  ```

  Placement Guidelines:
  - Consider real-world usage patterns
  - Respect vertical space constraints
  - Distribute objects logically across available surfaces
  - Leave some surfaces empty if appropriate, but include them as
  ↪ empty arrays

  Example:
  If layer_info shows dining table has layer_0 with surface_0 and
  ↪ surface_1:
  ```json
  {
    "dining table": {
      "layer_0": {
        "surface_0": [
          {
            "id": "place_setting_1",
            "position": [0.3, 0.3],
            "rotation": {"facing": "dining chair"},
            "rationale": "Positioned for the chair"
          }
        ],
        "surface_1": [
          {
            "id": "place_setting_2",
            "position": [-0.3, -0.3],
            "rotation": {"angle": 0},
            "rationale": "Opposite side placement"
          }
        ]
      }
    }
  }
  ```

small_objects_layered: >-
  You are required to populate only and exactly the following small
  ↪ objects """<SMALL_OBJECTS>"""
  on the following motif: """<MOTIF_DESCRIPTION>""",
  only on the following large furniture: """<LARGE_FURNITURE>""",
  in a room with description: """<ROOM_TYPE>""",

  The layer information is: """<LAYER_INFO>"""

  CRITICAL OBJECTIVE: You must place the EXACT total quantity
  ↪ specified for each object type across ALL furniture surfaces.
  ↪ This is a strict requirement - no more, no less.

  Quantity Distribution Strategy:
  1. First, identify all available surfaces across all layers
  2. Calculate how to distribute each object type to reach the exact
  ↪ total
```

```
  3. Ensure the sum of all amounts for each object type equals the
  ↪  required total exactly

  For each small object entry, provide:
    1. The name of the piece (use EXACTLY the same name as provided
    ↪  in the input)
    2. The appearance/style description of the piece (be specific,
    ↪  e.g. "glass cup")
    3. The dimensions of the piece [width, height, depth] in meters
    ↪  according to the description
    4. The amount of the piece (integer)

  JSON Structure Requirements:
    - Each large_object_name appears exactly once as a top-level key
    - Each layer_X appears exactly once under each large_object_name
    - Each surface_X appears exactly once under each layer_X
    - All layers and surfaces from the layer information must be
    ↪  included
    - CRITICAL: Do NOT duplicate layer keys (e.g., do not define
    ↪  "layer_0" multiple times)

  Additional Requirements:
    - Each entry must represent a SINGLE type of object, do not
    ↪  generate composite sets
    - If there are multiple small objects with different
    ↪  appearance/size, break them into multiple entries
    - Use exact number of layers and surface IDs from layer
    ↪  information
    - Consider space available on each surface and height of each
    ↪  layer (layer_0 is highest)

  Respond with JSON:
  ```json
  {
    "large_object_name": {
      "layer_0": {
        "surface_0": [
          {
            "name": "small_object_name",
            "description": "appearance/style description of a single
            ↪  small object (e.g. glass cup)",
            "dimensions": [width, height, depth],
            "amount": number of same small object (integer),
          },
        ]
      },
      "layer_1": {
        "surface_0": [
          {
            "name": "small_object_name",
            "description": "appearance/style description of a single
            ↪  small object",
            "dimensions": [width, height, depth],
            "amount": number of same small object (integer),
          },
        ]
      },
      ...
    },
    ...
  }
  ```

choose_objects: >-
  Choose from the following list of objects: """<OBJECT_LIST>"""
  Which objects usually has small objects placed on top/inside of it?

  You can respond with an empty array if it is absolutely certain
  ↪ that all objects are not meant to have any objects placed on
  ↪ top of them.
  Respond exactly with the object names in JSON format.
  ```json
  {"objects": ["object_1", "object_n", ...]}
  ```
```

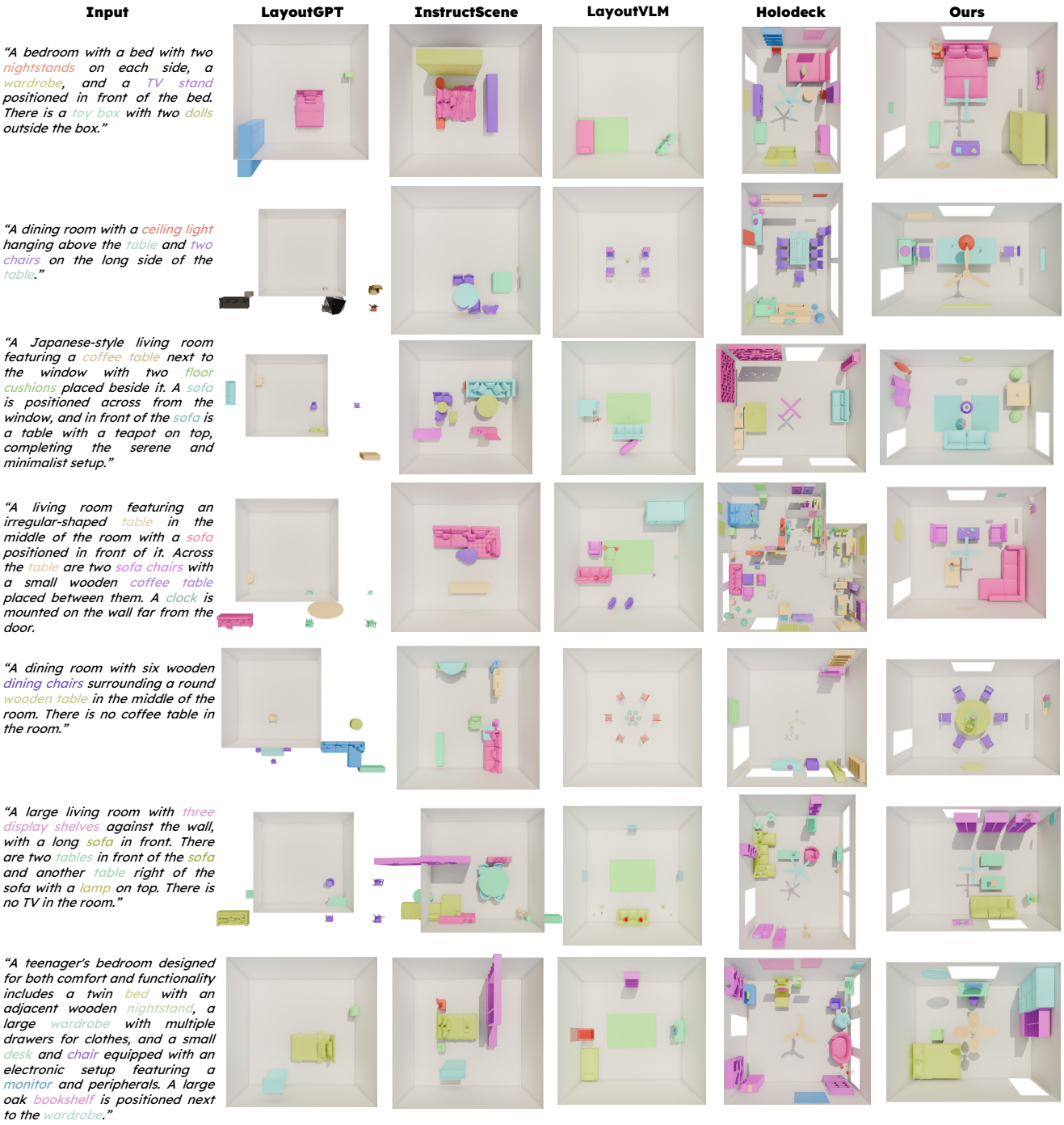

Figure 10. **Extra qualitative comparisons at the scene level.** Among all methods, HSM produces the most consistent room layouts and object arrangements with respect to the input descriptions.

Figure 11. **Rendered qualitative results.** HSM is able to generate realistic and densely populated scenes. The scenes also contain smaller objects and are aligned with user input.