**Model 1.2 🡪 Steps**

Okay, let's outline a clear architectural plan focusing on meeting the core requirements: generating floor plans based on **plot size, room counts, area considerations, and plausible room adjacencies/relationships.**

Given the need to handle both quantitative requirements (counts, size) and qualitative/relational aspects (adjacency, typical placement), a **Hybrid Architecture** appears most suitable. It breaks the problem down:

1. **Stage 1: Relational Structure Generation (Graph-based)**: Focuses on generating a *topologically plausible* layout graph that respects adjacencies and basic placement, conditioned on user requirements.
2. **Stage 2: Visual Rendering & Refinement (Conditional Image GAN)**: Focuses on translating the proposed structure into a detailed, visually coherent pixel-based segmentation map, ensuring walls align and shapes are well-formed.

Here is the detailed plan:

**Phase 1: Data Preparation & Preprocessing (CRUCIAL)**

1. **Image Dataset:** Your existing collection of PNG floor plans (5, 10, 20 Marla) with correct color coding.
2. **Metadata Extraction:**
   * Run the corrected generate\_metadata.py script.
   * **Output:** Per-image JSON files containing file\_name, plot\_type, dimensions, room\_counts, room\_areas\_pixels. This provides the explicit requirements associated with each ground truth image.
3. **Graph Extraction (Complex Step):**
   * **Goal:** Convert each ground truth segmentation map image into a corresponding **graph representation**.
   * **Process:**
     + Use image processing (e.g., OpenCV connectedComponentsWithStats) on each color channel (excluding walls/background) to identify distinct room instances and their pixel areas/bounding boxes.
     + Determine **adjacency**: Analyze boundaries between detected room components. If two rooms share a significant boundary (not just a corner, potentially excluding wall pixels), create an edge between their corresponding nodes in the graph. Store adjacency info.
     + Identify **connectivity**: Detect doors (using the 'Door' color) or openings between adjacent rooms and potentially add edge attributes representing these connections.
     + **Graph Structure:**
       - *Nodes:* Represent individual rooms/spaces (e.g., Bedroom\_1, Bathroom\_1, Kitchen\_1, Lounge\_1, Garage\_1, Stairs\_1, Lawn\_1).
       - *Node Attributes:* room\_type (string/enum), pixel\_area (int, from image), bounding\_box (tuple, optional), plot\_type (string).
       - *Edges:* Represent adjacency/connectivity between rooms. Edge attributes could indicate connection\_type (e.g., 'adjacent\_wall', 'door', 'opening').
   * **Output:** A dataset where each entry links: image\_filename -> metadata (JSON) -> layout\_graph (e.g., using libraries like NetworkX to store/serialize).
   * **Tooling:** Requires significant Python scripting with OpenCV, potentially NetworkX. Accuracy here is key.

**Phase 2: Stage 1 Model - Graph Generation Network (e.g., Conditional Graph VAE or GAN)**

1. **Input:** Encoded User Requirements:
   * plot\_type: One-hot encoded vector.
   * room\_counts: Vector of target counts for each relevant room type (e.g., [num\_bedrooms, num\_bathrooms, num\_kitchens, ...]).
   * (Optional) Area constraints/preferences.
2. **Architecture:** A model designed for conditional graph generation. Options:
   * **GraphVAE:** Learns a latent space of graphs. Encoder maps ground truth graphs to latent vectors. Decoder generates graphs from latent vectors, conditioned on requirements.
   * **GraphGAN:** Generator produces graphs based on requirements + noise. Discriminator learns to distinguish real vs. generated graphs, also conditioned on requirements.
   * **Autoregressive Graph Model:** Generates nodes and edges sequentially based on requirements and partial graph.
3. **Output:** A proposed layout\_graph (nodes, edges, attributes) that satisfies the input requirements and reflects learned adjacency patterns.
4. **Training:**
   * **Data:** Pairs of (Encoded Requirements from Metadata, Ground Truth Layout Graph from Preprocessing).
   * **Loss:** Depends on architecture (e.g., VAE reconstruction + KL divergence, GAN adversarial loss, graph structure comparison losses). Aims to generate graphs structurally similar to real ones meeting the same requirements.
5. **Libraries:** PyTorch Geometric (PyG) or Deep Graph Library (DGL) are commonly used for GNNs.

**Phase 3: Stage 2 Model - Conditional Image GAN (Layout Refinement)**

1. **Input:**
   * **Rasterized Graph Proposal:** A simplified 2D image generated from the layout\_graph produced by Stage 1. This map shows the approximate placement, size, and type of each room (e.g., colored blobs/rectangles based on node attributes and rough topology). Needs a consistent rasterization function (graph\_to\_raster).
   * **Encoded User Requirements:** The same requirement vectors used for Stage 1 (plot type, room counts) are fed as additional conditions to ensure consistency and guide refinement.
2. **Architecture:** A Conditional GAN suitable for image generation, likely U-Net based.
   * **Generator:** Takes the rasterized proposal map and requirement conditions as input. Outputs the final detailed segmentation map (e.g., 256x256xNumClasses logits). Similar to the Pix2Pix generator structure, but input is the rasterized map, not a real image or noise.
   * **Discriminator:** Takes the (real or generated) detailed segmentation map and the requirement conditions. Outputs a patch-based realism score (PatchGAN).
3. **Output:** Final segmentation map (pixel grid) with detailed room shapes, aligned walls, etc.
4. **Training:**
   * **Data:** Pairs of ((Rasterized *Ground Truth* Graph + Encoded Requirements), Ground Truth Segmentation Map Image).
   * **Loss:** Standard cGAN losses (Pixel-wise loss like Sparse Categorical Crossentropy between generated map logits and ground truth map indices + Adversarial loss).
5. **Libraries:** TensorFlow/Keras or PyTorch.

**Phase 4: Inference Workflow (Generating for User)**

1. User inputs requirements (e.g., "10 Marla, 2 Bedrooms, 2 Bathrooms, 1 Kitchen...").
2. Encode these requirements into the format expected by Stage 1.
3. Pass encoded requirements to the trained **Stage 1 Graph Generator** -> Get a proposed layout\_graph.
4. Execute the graph\_to\_raster function on the proposed graph -> Get the rasterized proposal map image.
5. Pass the rasterized map + encoded requirements to the trained **Stage 2 Image GAN Generator** -> Get the final segmentation map (logits).
6. Convert logits to class indices (argmax).
7. Convert class index map to an RGB image using INDEX\_TO\_COLOR mapping.
8. Save/display the final RGB PNG image.
9. (Future) Develop function to convert the final segmentation map to DXF.

**Key Advantages of this Hybrid Plan:**

* **Separation of Concerns:** Stage 1 handles complex relational logic; Stage 2 handles detailed visual rendering.
* **Leverages Strengths:** Uses GNNs for what they are good at (relationships) and GANs for what they excel at (image generation/refinement).
* **Direct Conditioning:** Both stages are directly conditioned on user requirements.
* **Implicit Rules:** The GNN can learn implicit adjacency/placement rules from the graph structures in the data.

**Major Challenges:**

* **Graph Extraction:** Developing the algorithm to reliably convert images to graphs is the most significant preprocessing challenge.
* **Rasterization:** Defining an effective way to visually represent the graph for Stage 2.
* **Training Complexity:** Training two generative models requires careful tuning.

This plan provides a clear, albeit complex, path toward building a system that can generate floor plans respecting both quantitative counts and qualitative relational structures based on user input.