ChatGPT

Integrating a Lightweight Diffusion Transformer (DiT) into PaCo for Point Cloud Completion

Retaining PaCo’s Parametric Plane Paradigm

PaCo’s key innovation is its **parametric completion** pipeline, which represents surfaces as plane primitives rather than dense points [1] [2]. This paradigm must be **preserved**: the final completion is still given by a set of plane parameters with inlier points (plane “proxies”), enabling direct assembly into a polygonal mesh. In practice, this means we keep **all the original PaCo modules for parametric recovery** – the plane **parameter estimator**, point **distributor**, and primitive **selector** – unchanged [3] [4]. The **only modification** will be in the global encoding stage (the “proxy generation” stage), where we replace PaCo’s original PCTransformer with a Diffusion Transformer (DiT) module. By doing so, we ensure the **plane-proxy representation and parameter estimation remain the core** of the completion process, maintaining PaCo’s advantages in producing compact, structured outputs [5] [6].

Lightweight DiT for Global Point Cloud Encoding

We propose to replace PaCo’s **PCTransformer-based proxy generation** module with a **lightweight Diffusion Transformer (DiT)** encoder/decoder (approximately an 8-layer transformer) for global feature encoding and missing-plane generation. In the original PaCo (illustrated in **Figure 2**), an incomplete point cloud is first segmented into plane patches and encoded into a set of plane proxy features $V={v_i}$ (one per detected partial plane) [7] [8]. These plane proxies serve as context, and PaCo’s PCTransformer then attends to them with a set of learned query vectors (proposals for missing primitives) to produce new plane proxies completing the object [9] [10]. We will **retain this overall logic**, but implement it with a DiT:

Architecture of the original PaCo model [11] [12]. We will replace the “Transformer Decoder” (PCTransformer) in the Proxy Generation stage with a Diffusion Transformer module, while keeping the parametric recovery heads unchanged.

DiT Encoder–Decoder Design: We use the incomplete cloud’s plane proxies V as **conditioning context** and introduce a set of M **latent query tokens** that represent the to-be-completed planes. In PaCo, M queries (from both existing and generated sets) are chosen via a ranking strategy [10]; similarly, we can initialize our latent queries to either (a) the encoded features of known planes (for planes that are partially present) and placeholders for missing ones, or (b) simply a learned set of M **trainable query embeddings** (akin to DETR-style object queries [13]) that will query the context for new planes. These query tokens will pass through an **8-layer Diffusion Transformer** decoder: each layer applies self-attention among the query tokens and cross-attention between query tokens and the context plane proxies V. In this way, the DiT **extracts a global feature encoding of the partial point cloud and infers the missing structures** by attending to all existing planar regions simultaneously. The final output of the DiT decoder is a set of **proposed plane proxies** (each a feature vector) that collectively represent the completed object’s surfaces, which are then fed into the unchanged parametric heads (predicting plane equations and inlier point distributions) [14] [15].

Fewer Parameters, Comparable Performance: By leveraging the powerful self-attention mechanism of transformers, an 8-layer DiT can capture long-range dependencies and global geometry **even with a relatively small number of parameters**. In image diffusion, replacing a heavy U-Net with a transformer backbone has been shown to preserve generative performance without inductive biases [16] [17]. Here, our DiT (with ~8 layers) can likely **match or exceed the original PCTransformer's completion ability** while using fewer parameters and layers. The original PCTransformer from PoinTr was a dedicated point cloud transformer network; the DiT, by contrast, benefits from **rich conditioning (time embeddings, see below) and iterative refinement ability**, potentially enabling better inference of missing planes from incomplete data. In summary, the DiT module serves as a drop-in replacement for PCTransformer to produce global point-cloud encodings, but with a leaner architecture that focuses model capacity on **global context modeling** rather than large per-point or per-patch convolutions. This helps maintain PaCo's strong performance in completing large missing regions [18] while improving efficiency.

Enhancing Detail Completion with Hierarchical Diffusion and Multi-Scale Features

One challenge in point cloud completion is recovering **fine details** – small structures or intricate boundaries – especially from very sparse or occluded inputs. We propose several mechanisms to augment the DiT module to better capture local topology and fine-grained geometry:

- **Hierarchical Diffusion Process:** Rather than generating all missing geometry in one shot, we can employ a **multi-step diffusion strategy** to progressively refine the completion. During training, we treat the set of ground-truth plane proxies (for the complete shape) as the target and simulate a **denoising diffusion** process: we add noise to the target plane embeddings and train the DiT to **denoise** them conditioned on the partial input. Over T timesteps, the model learns to gradually "fill in" the missing information. At inference, we can start from pure noise (random plane tokens) and run the DiT iteratively for T steps, guided by the partial cloud, to sample a set of completed plane proxies. This **iterative refinement** allows large structures to form first and small details to emerge later – analogous to how diffusion models generate images from coarse to fine. A similar idea was successfully used in recent point-cloud completion work: e.g. Lyu et al. generate a **coarse point cloud with a conditional DDPM and then refine it** with a second network [19]. In our case, the first diffusion stages of the DiT would focus on adding major missing planes, while later stages resolve finer details like small planar patches or adjust plane boundaries for consistency. Notably, we can keep the DiT **lightweight (8 layers)** by reusing it across diffusion steps, instead of using a deeper one-shot network. This hierarchical diffusion approach should improve the model's ability to **hallucinate plausible structures** in severely incomplete regions while preserving global coherence.

- **Multi-Scale Feature Fusion:** We will incorporate a **multi-scale encoding** of the point cloud to feed into the DiT. PaCo already builds a hierarchy (point → patch → plane) [7] [20]. We propose to enhance this by giving the DiT access to both **coarse and fine features**. For example, alongside each plane proxy (which is a pooled representation of an entire segmented patch), we could attach or attend to the finer **point-level or patch-level features** that make up that plane. Concretely, the DiT's cross-attention could attend not just to a single token per plane, but also a set of sub-tokens representing smaller local neighborhoods or edges on that plane. This multi-scale attention ensures that while the DiT focuses on global plane structures, it also receives signals about **local topology** (e.g. curvature along an edge, small holes or extrusions on a plane). We can implement this by extending the context: let $V = {v_i}$ be plane tokens as before, and let ${p_j}$ be a set of **point/patch tokens** (e.g. the original point proxies or patch embeddings from the hierarchical encoder [21]). The DiT can alternate attention between plane-level context and

point-level context, or use separate cross-attention heads for each scale. By **fusing multi-scale features**, the model should better reconstruct small structures that might be lost in a purely plane-level representation. Another approach is a two-stage completion: first have the DiT predict coarse plane proxies (perhaps with fewer tokens or at lower resolution), then have a **refinement network or second DiT** that takes those coarse results and the original input to predict finer details (e.g. splitting planes or adding missing small planes). This two-stage design mirrors a coarse-to-fine strategy and can significantly improve fine detail completion [19].

- **Edge-Preserving Enhancements:** In polygonal models, sharp edges occur at the intersection of planes and should be preserved during completion. To encourage this, we can introduce mechanisms in the DiT (or in the loss functions) that focus on **boundary conditions**. One idea is to provide the DiT with explicit information about observed edges in the partial input. For instance, we can preprocess the incomplete point cloud to detect boundary points (points that lie at the edge of a plane segment). These can be encoded as special **edge tokens or features** fed into the transformer. The DiT could use cross-attention to these edge tokens to ensure that predicted planes align well with existing boundaries and that new planes meet neatly at likely edge locations. Additionally, we can modify the **diffusion objective or loss** to be edge-aware: for example, when comparing the completed output to ground truth, weight errors along primitive boundaries more heavily (so the model learns to get edges right). Another strategy is to enforce **planar intersection consistency**: during training, if two predicted planes in the output should intersect along an edge (as per ground truth), we penalize deviations in their intersection line or angle. This kind of constraint could be added as an auxiliary loss encouraging the DiT to output planes that form clean edges. Finally, a **small post-processing refinement** could be applied: after obtaining plane parameters, we could snap or adjust adjacent planes so that their lines of intersection are crisp (this could be guided by a fitted edge from input data). By integrating these edge-preserving ideas, the completion will better retain sharp corners and small structures (like thin slabs or beams) that are crucial for visual and structural fidelity.

In summary, these enhancements leverage the flexibility of the Diffusion Transformer to incorporate **coarse-to-fine generation, multiple feature scales, and geometric constraints**, all aimed at improving the restoration of local topology and fine details in the completed point cloud.

Performance vs. Training Efficiency Trade-Offs

Introducing a diffusion-based transformer and multi-scale mechanisms affects both performance and efficiency, so we must analyze the trade-offs:

- **Model Complexity vs. Quality:** The original PCTransformer module was a one-pass transformer (or a small stack) producing outputs in a single forward pass [22]. Our DiT approach, especially if using iterative diffusion steps or a two-stage coarse-to-fine process, will increase computation. For example, a 100-step DDPM would require 100 forward passes of the network to sample a completion, which is computationally expensive [23]. However, running the full 1000-step diffusion may not be necessary for good results – prior work has shown that **accelerated diffusion with fewer steps (e.g. 50)** can still produce reasonable coarse completions, which can then be refined [24] [25]. We can adopt similar strategies: use a moderate number of diffusion steps (e.g. 20–50) or a clever **scheduler** to reduce inference time, and rely on the refinement or parametric head to polish the result. We might also explore **Diffusion distillation** or **denoiser acceleration** techniques from the literature to cut down the number of steps at inference (turning a many-step diffusion into a few large denoising leaps). Each step of our DiT is only 8 layers (much lighter than a typical U-Net), so even multiple steps might be acceptable – but we will need to

profile and perhaps choose a trade-off (e.g. 10 steps for most cases, which could still significantly improve detail over a single pass).

- **Parameter Count vs. Coverage:** Using an 8-layer DiT likely **reduces the parameter count** compared to PaCo's original transformer (especially if that included heavy MLPs or multiple layers for query generation and self-attention). This is beneficial for memory and may allow training on larger batches or higher resolution data. However, a smaller model might struggle if asked to model extremely complex shape variations. We anticipate that by leveraging the diffusion mechanism (which effectively allows repeated application of the same network for iterative improvement), we **mitigate the need for a very deep network** – the model can apply itself multiple times to refine the output. This iterative reuse is parameter-efficient (like an RNN reusing weights over time). Thus, our approach shifts some complexity from "more parameters" to "more computation" (more iterations). If training time becomes an issue, one could initially train the model with fewer diffusion steps (or even as a single-shot transformer) to get a reasonable starting point, and then gradually increase the diffusion iterations or introduce fine-refinement stage later (a form of curriculum that first learns coarse completion, then fine completion as capacity allows).

- **Training Complexity:** Training a diffusion model typically involves sampling random timesteps and optimizing an expected loss over noise levels, which can be slower than standard supervised training. In our conditional setting, each training iteration will sample a timestep t and add noise to the ground-truth plane proxies, then run the DiT to predict either the original clean proxies or the noise (depending on training objective). This is more involved than PaCo's original training (which directly supervised predicted vs. true primitives via set matching [4] [26]). To keep training efficient, we can adopt techniques like **reparameterization** (predict noise ϵ instead of denoised signal) and use **loss weighting** (e.g. weight losses by the noise level schedule) as in standard DDPM training. We might also limit the number of diffusion timesteps considered during training – e.g., train with a subset of timesteps (as done in some accelerated diffusion methods) or use a **progressive training** approach (first train model to handle small noise, then gradually increase). Fortunately, our domain (shape completion) might not require as many timesteps as high-resolution image synthesis; we can likely use a coarse noise schedule (say $T=50$ or 100 total) to cover the range from totally missing to fully detailed. This balances training time with outcome quality. We should also consider **mixed precision** and other training optimizations given the transformer nature of the model, to further alleviate the overhead of diffusion training.

- **Memory and Throughput:** The use of multi-scale attention and edge tokens increases the number of tokens the DiT processes (compared to just one token per plane). We'll need to manage this carefully to avoid memory blow-up, especially for scenes with many small planar segments. One optimization is to restrict multi-scale processing to necessary cases: e.g., perhaps only use detailed point-level tokens for regions identified as tricky (like high curvature or containing small structures), and not for large simple planes. Another idea is a **sparsity mechanism** in attention: since each plane token might only need to attend strongly to its own constituent point tokens and a few neighboring planes' tokens, we could use a localized attention mask or two-step attention (first plane-to-own-points, then plane-to-plane) rather than full dense attention over all tokens. This would improve speed and memory. We can also keep the embedding dimensions moderate (e.g. 256 or 384) to control memory usage; an 8-layer transformer with such width is quite tractable.

In conclusion, our design tries to **shift complexity into an iterative process** rather than a monolithic network. This yields a flexible quality-efficiency trade-off: with fewer steps, the model is faster but might

miss some fine details; with more steps or an added refinement pass, we approach higher completeness and accuracy at the cost of time. We will experimentally find a sweet spot where the completion quality is significantly improved (especially for highly incomplete inputs) while training/inference remain feasible. Techniques like model distillation or step-reduction can further push this balance towards efficiency if needed [24] [25].

Adapting the DiT Architecture to Sparse Point Clouds

Integrating a DiT originally designed for images into a sparse point cloud scenario requires some architectural customizations:

- **Patch/Token Design:** In image DiTs (Diffusion Transformers for images), the input is patchified 2D pixels [27]. For point clouds, we lack a regular grid, so we define "patches" differently. PaCo's pipeline conveniently gives us **plane proxies** which act as **tokens** representing large planar patches of the shape. We will use each plane proxy v_i (which encodes all points on a partial plane) as a **token in the transformer's context sequence**. This is akin to treating each detected plane segment as a "super-patch." Additionally, as mentioned, we may include smaller-scale tokens such as point cluster features or edge features. These can be treated as additional tokens in the sequence. To differentiate token types (plane-level vs point-level vs edge), we can add a **type embedding** to each token (a learned vector indicating "this token represents a whole plane" vs "this is a local patch/edge"). Position encoding in point clouds is tricky since there is no canonical ordering, but we can encode **geometric attributes** of each plane as a pseudo-position: for example, use the plane's centroid coordinates or its normal vector as part of a positional embedding. One approach is to encode each token with a triplet (x,y,z) position (centroid) and perhaps orientation (like the normal $\mathbf{n}$) via an MLP to a positional encoding, which is then added to the token feature. This gives the transformer some sense of spatial arrangement akin to 2D positional embeddings [28]. For local point patch tokens, a similar idea can be used (each patch token knows its center coordinates). By designing the token embeddings this way, the DiT can better reason about spatial relationships between tokens (e.g. which planes are adjacent or far apart) even though the tokens are an unordered set.

- **Time Embedding for Diffusion:** In diffusion models, a **time (timestep) embedding** is injected to inform the network of the current noise level. We will incorporate a time embedding in the DiT architecture, especially if we train it as a denoiser. A common method is to use a sinusoidal positional encoding of the timestep or learn an MLP that maps the scalar t to an embedding vector, then use this to modulate the network. For example, we can follow the approach of Diffusion Transformers in vision: apply the time embedding as a bias or scale in each transformer block via **adaptive layer normalization (AdaLN)** [29]. Concretely, each transformer layer's normalization can be **conditioned** on the time embedding by shifting and scaling the normalized activations (this injects t into every layer). Alternatively, we could concatenate the time embedding to each token or add it to the token features (though for a set of tokens, AdaLN or a FiLM-style modulation per layer is more controlled). If we are not doing multi-step diffusion (e.g. using DiT in a single forward pass deterministically), we might not need a time embedding at all (that would reduce to a standard conditional transformer). But given we plan to leverage some diffusion steps, the time embedding is crucial to train the model to handle varying noise levels. We will likely use a standard sinusoidal encoding (as in DDPM) or train a small Fourier-feature MLP for t, and incorporate it via AdaLN in each transformer block [30]. This will allow the model to **adapt its behavior**: e.g., at high-noise (early steps) focus on broad structure, at low-noise (later steps) focus on fine detail – exactly the behavior we want.

- **Conditional Inputs and Cross-Attention:** Conditioning on the partial input is central to our design. In DiT for images, one might feed conditional information via **cross-attention** (e.g. attending to text embeddings for text-to-image models) [31] or via extra input tokens [32]. We will implement **cross-attention layers** in the DiT that allow the incomplete-cloud context to influence the output tokens. Specifically, our DiT decoder layers will alternate between self-attention on the query tokens and cross-attention from query tokens to the context tokens V (the plane proxies of the input). This is analogous to the transformer decoder in PaCo/Detr [22], but spread across multiple layers. Each query token can attend to all plane proxy features, enabling it to "see" which parts of the object are already present and which are missing. By stacking multiple such layers, the queries iteratively refine their understanding of what new planes are needed where. We may also consider **encoder-side conditioning**: for instance, first use a small **transformer encoder** on the input plane proxies V themselves to produce a contextualized representation of the partial shape. (If the plane proxies are many, this helps them share info among themselves – e.g., multiple partial planes that are co-planar might get merged cues.) The encoded V would then be used as keys/values for the DiT decoder. This essentially reproduces a **encoder-decoder architecture** (with V as encoder tokens, queries as decoder tokens), which is a natural fit for conditional generation. Another possible conditional pathway is through **Adaptive Layer Norm** as mentioned: we could encode the entire partial point cloud into one global feature (say by pooling V or using a separate point encoder like PointNet) and use that global feature to modulate the DiT layers (similar to how class labels or text condition can modulate a diffusion model). However, given that we have a rich token-wise representation of the input already, cross-attention is a more fine-grained and direct way to condition on the input geometry. We will thus focus on a cross-attention based conditioning at each DiT layer, possibly combined with global conditioning in layer norms. This dual conditioning ensures the DiT outputs remain **grounded in the input** – the generated planes align with or extend from actual observed surfaces, rather than floating off in inconsistent ways.

- **Handling Sparsity:** Point clouds (especially partial ones) are sparse and irregular. The plane-proxy representation partially addresses this by compressing many points into one proxy. Still, some incomplete inputs may have very few points or planes (e.g., only a couple of partial surfaces visible). The DiT must handle extreme cases robustly. One adaptation is to make the model robust to varying token counts. We will use padding and masking for the context tokens V (PaCo already pads V to a fixed size [13]) so that the transformer can process a variable number of detected planes. We also pad the query tokens to a fixed M (the maximum number of planes we allow the model to predict). During training, if an object has fewer ground-truth planes than M, we'll treat the extra query outputs as "empty" and not penalize them (similar to PaCo's set matching with padded null primitives [4] [33]). This ensures the model doesn't break when the partial input is very sparse or when the completed model should have fewer planes than the upper bound. Additionally, to cope with **noise in sparse data**, we might incorporate augmentation like jittering the input points or varying the dropout of context tokens during training – forcing the DiT to learn to complete even when some context planes are missing or uncertain. Such training could improve robustness to real-world sensor noise or severe occlusions.

In essence, these architectural adaptations repurpose the DiT to the point cloud domain: we replace image patches with plane/point tokens, include appropriate spatial embeddings, integrate diffusion time conditioning, and implement strong cross-modal (partial-to-complete) attention. With these changes, the DiT will be well-tailored to **sparse, unordered 3D data**, effectively bridging the gap between diffusion models and PaCo's parametric completion.

Training Strategy and Workflow

Finally, we outline a feasible **training strategy** to integrate the DiT module into PaCo:

1. **Pretraining or Initialization:** We may begin by initializing parts of the model with known solutions. For instance, the hierarchical encoder (point patches to plane proxies) can be initialized from the original PaCo or PoinTr weights [21], ensuring the plane proxy features are meaningful from the start. The DiT itself could benefit from initialization too – if there are any pre-trained Diffusion Transformers for 3D or even 2D that make sense (perhaps a ViT pre-trained on image patches), those weights might be adapted. However, given the novelty of our task, it's likely we train DiT from scratch. We will at least initialize the time embedding modules (if any) with standard sin/cos functions and the query tokens (if using learned queries) with small random vectors. If training from scratch proves difficult, an alternative is to **first train the DiT in a deterministic mode**: disable the diffusion noise, and train it to directly predict plane proxies from the partial input in one pass. This essentially replicates the original PaCo training (supervising the output plane set against ground truth using set matching and the losses for plane parameters, point alignment, etc. [26] [34]). Once this is working (i.e., the DiT can replace PCTransformer in a one-shot manner), we can **gradually introduce diffusion**: start adding small noise to the input queries during training so the model learns to refine noisy proposals. Over time, move to the full diffusion objective where we randomize the noise level t. This curriculum (from one-shot to diffusion) can stabilize training, ensuring the model first understands completion before tackling denoising tasks.

2. **Loss Functions:** We will use a combination of **diffusion losses and PaCo's parametric losses**. For diffusion, the typical loss is an L_2 predicting the added noise (for a given timestep) or predicting the clean data. In our case, the data is the set of ground-truth plane proxy features. However, those features are not given; what we do have are the ground-truth plane parameters and inlier points. We can obtain a proxy feature for each ground-truth plane by encoding its points through the same encoder (treating the ground truth complete cloud as if it went through PaCo's encoder) – effectively we get a target embedding for each true plane. The diffusion model can then be asked to predict that embedding. An easier approach is to train the diffusion model indirectly via the parametric output: have the DiT output plane proxies and then apply the parametric heads and compute losses in the output space of plane parameters and point distributions. For example, after T diffusion steps (final output), we can apply the standard PaCo losses: plane normal regression loss, Chamfer distance on distributed points, primitive selection loss, etc. [26] [34]. These will guide the model to produce correct plane proxies. We might also supervise intermediate diffusion steps for stability: e.g., ensure that after a certain number of steps the partial shape is at least reconstructed (a form of intermediate reconstruction loss). If training with the **denoising score matching** objective, we will incorporate that as well: for a sampled timestep t, we add noise to ground-truth proxies and have the DiT predict the noise (or denoised signal) – using a plain L_2 loss on the feature vectors. This encourages the DiT to properly perform denoising at each step. In summary, our loss could be a **weighted sum** of: diffusion noise prediction loss (averaged over timesteps), plus final output losses from PaCo (which themselves include several terms on plane params, point positions, uniformity, and selection [35] [34]). The set matching procedure from PaCo will still be used to align predicted plane proxies with ground truth planes (this can be done at final output; for diffusion intermediate steps, set matching is less meaningful because those are not final predictions). By combining these losses, we train the DiT to not only denoise in feature space but also to output features that correspond to correct geometric primitives.

3. **Training Regime:** We will likely train this model end-to-end, feeding an incomplete point cloud, running through the encoder, DiT, and parametric heads, and applying the losses. Key considerations are **batch size and learning rate**: transformers can be data-hungry, and diffusion models often benefit from lower learning rates and many epochs. Given PaCo's training was 600 epochs on ABC dataset [36] [37], we might aim for a similar or slightly longer schedule, especially if diffusion noise makes convergence slower. We will use **gradient checkpointing** for the DiT if needed to handle memory (since each sample might go through multiple diffusion steps in one forward if we unroll, but more likely we sample one t per sample per iteration, which is standard and memory-friendly). We also ensure the **masking of padded tokens** is correct during training, so that varying number of planes doesn't cause issues in loss computation. For diffusion, we'll train with the **whole range of t uniformly** (or cosine-weighted as some do) to make sure the model learns all levels of refinement. As training progresses, we should monitor not just final Chamfer distance and normal consistency (as PaCo did) but also possibly the model's performance at intermediate steps (to ensure the diffusion is working). If we see issues (like model only learns to output something meaningful at $t=0$ but not improve gradually), we might add **auxiliary supervision**: e.g., after a certain number of diffusion steps, decode and supervise against a partially completed ground truth. For example, after half the steps, the model should have recovered all larger planes – we can obtain a "halfway" target by taking the ground truth and removing the smallest primitives or something. This is optional and more complex, but could reinforce the hierarchical aspect.

4. **Inference Strategy:** Once trained, we have choices for inference depending on desired speed/quality. We could run the full diffusion iterative process: start with M random query tokens (or a learned average embedding plus noise), and iterate the DiT update T times to sample a set of completed plane proxies. Each iteration involves cross-attending to the fixed partial input proxies (which we encode once at start). We can incorporate **classifier-free guidance** if needed (though in our case "guidance" would mean pushing the output towards more likely shapes – we might not need this explicitly). If T is large, we can adopt the **accelerated sampling** (like using larger step size or DDIM deterministic sampling) to reduce steps at some quality cost [24]. In scenarios where speed is critical, an alternative is to use the DiT in a single pass by directly feeding $t=0$ (no noise) and relying on its learned one-step capability (essentially falling back to a deterministic completion like the original PaCo). The model was trained to handle all t, so at $t=0$ it should produce a plausible completion. This gives very fast inference (just one forward pass), likely with slightly less detail than multi-step. We could also explore a **hybrid**: e.g., do 5 diffusion steps to get coarse structure, then one big jump to $t=0$ to finalize – leveraging a bit of iteration for quality, but far fewer steps than a full chain.

Through this training and inference setup, we aim to harness the strengths of both worlds: PaCo's parametric accuracy and Diffusion Transformer's generative completeness. The output will remain a set of planar primitives that can be directly evaluated with PaCo's metrics (Chamfer, normal consistency, etc.), and we anticipate improved performance especially in cases of high missing data (as the generative nature of diffusion can fill in plausible geometry where deterministic methods struggle). The training process is admittedly more involved, but is made feasible by the lightweight nature of the DiT (8-layer transformer) and the clever combination of objectives to guide it. Overall, this integrated strategy should yield a new state-of-the-art in parametric point cloud completion – maintaining PaCo's **"parameter-plane" core** while infusing it with the powerful **diffusive generation capability** for enhanced global reasoning and detail recovery.

References: PaCo architecture and parametric completion paradigm [7] [13]; Diffusion Transformer design for conditional generation [29] [38]; Conditional diffusion for point cloud completion [19].

---

1 2 3 4 5 6 7 8 9 18 20 21 26 33 34 35 36 37 [2503.08363] Parametric Point Cloud Completion for Polygonal Surface Reconstruction
https://ar5iv.org/abs/2503.08363

10 11 12 13 14 15 22 Parametric Point Cloud Completion for Polygonal Surface Reconstruction
https://openaccess.thecvf.com/content/CVPR2025/papers/Chen_Parametric_Point_Cloud_Completion_for_Polygonal_Surface_Reconstruction_CVPR_2025_paper.pdf

16 17 27 28 29 30 31 32 38 Scalability of Diffusion Models with Transformer Backbone | Encord
https://encord.com/blog/diffusion-models-with-transformers/

19 23 24 25 GitHub - ZhaoyangLyu/Point_Diffusion_Refinement: A Conditional Point Diffusion-Refinement Paradigm for 3D Point Cloud Completion
https://github.com/ZhaoyangLyu/Point_Diffusion_Refinement