NERFIFY: A Multi-Agent Framework for Turning NeRF Papers into Code

Supplementary Material

Summary of Supplementary Material

1. Extended Experimental Analysis

1.1. Complete NERFIFY-BENCH Results

1.1.1. Quantitative Metrics for Set 1 (Never-Implemented Papers)

Paper	Reported			Human Impl.			NERFIFY (Ours)		
	PSNR↑	SSIM↑	LPIPS↓	PSNR↑	SSIM↑	LPIPS↓	PSNR↑	SSIM↑	LPIPS↓
KeyNeRF [21]	25.65	0.89	0.11	25.70	0.89	0.12	26.12	0.90	0.09
mi-MLP NerF [34]	24.70	0.89	0.09	22.64	0.87	0.15	22.85	0.87	0.15
ERS [24]	27.85	0.94	0.06	26.87	0.90	0.12	27.02	0.90	0.12
TVNeRF [33]	27.44	0.93	0.08	26.81	0.92	0.12	27.30	0.92	0.10
Anisotropic NeRF [27]	34.08	0.97	0.05	28.85	0.94	0.06	29.01	0.94	0.06
NeRF-ID [1]	25.15	0.94	-	23.01	0.89	0.13	23.10	0.89	0.13
Surface Sam.. [32]	25.63	-	-	24.35	0.90	0.11	24.40	0.90	0.11
LiNeRF [4]	25.60	0.93	0.08	23.30	0.90	0.14	23.32	0.89	0.14
HybNeRF [28]	33.94	0.96	0.047	30.45	0.94	0.07	30.51	0.95	0.07
AR-NeRF	20.36	0.79	0.17	19.00	0.76	0.19	20.05	0.78	0.18

Table 1. **Comparison of NERFIFY with paper and human implementations.** We evaluate NeRF papers from the NERFIFY-BENCH set whose code is not publicly available, using SSIM, PSNR, and LPIPS metrics. **Note.** Other baselines like Paper2Code, AutoP2C, GPT-5 and R1 failed to generate trainable code.

1.1.2. Quantitative Metrics for Set 2 and Set 3 (Github Repository Available)

Method	Original Repository			NERFIFY		
	PSNR ↑	SSIM ↑	LPIPS ↓	PSNR ↑	SSIM ↑	LPIPS ↓
Vanilla NeRF [16]	31.36	0.95	0.04	31.36	0.95	0.04
Nerfacto [25]	20.36	0.82	0.22	20.36	0.82	0.22
SeaThru-NeRF	27.89	0.83	0.22	30.08	0.92	0.07
BioNeRF	25.66	0.93	0.05	23.75	0.90	0.11
InstantNGP	32.77	-	-	32.64	0.96	0.06
ℓ_0 Sampler [14]	29.21	-	0.04	30.13	0.97	0.03
InfoNeRF [11]	18.27	0.81	0.23	17.87	0.69	0.44
DeblurNeRF	32.08	0.93	0.5	31.10	0.86	0.06
NerfingMVS	31.43	0.96	-	31.51	0.92	0.09

Table 2. **Comparison with existing implementations**. Evaluation of NERFIFY against original author repositories or gold-standard implementations.

Configuration	PSNR	SSIM	LPIPS
NERFIFY (Full)	27.16	0.91	0.11
Knowledge Sources: w/o Citation Recovery and In-context Examples	23.22	0.87	0.26
Validation & Feedback: w/o VLM Feedback and Smoke Testing (Stage 4)	9.39	0.80	0.42
Planning Strategy: One-Shot (no GoT) (Stage 3)	24.52	0.90	0.16

Table 3. **Component ablation study**. We evaluate the impact of each system component on synthesis quality and efficiency. Numbers are averaged on trainable implementations among 10 NERFIFY-BENCH Set 1 papers.

1.2. Qualitative Comparisons Gallery

1.2.1. Side-by-side Visual Comparisons

Figure 1 and Figure 2 shows the qualitative comparison of some of the methods in Set 1 of the NERFIFY bench.

1.3. Additional Ablation Studies

The ablation study experiments with the image quality metrics are shown in Table 3.

2. Comparison with Multi-Agent Baselines

Table 4 compares NERFIFY with automated coding systems on papers from
nerfifybench evaluating executability and semantic score. Recent PaperBench results show DeepCode (75.9% accuracy) substantially outperforms PaperCoder (51.1%), Claude Code (58.7%) and LLM-based systems like o1 (43.3%) [9]. So, we use DeepCode (with gpt-5 api) as our baseline alongside multi-agent frameworks GPT 5 thinking [20], ChatDev [23] (with gpt-5 api) and MetaGPT [10] (with gpt-5 api).

Table 4. Comparison of code generation systems on NeRF papers (in-depth comparisons shown in supplementary).

Paper	GPT-5 Thinking		ChatDev		MetaGPT		DeepCode		NERFIFY	
	Score	Trainable	Score	Trainable	Score	Trainable	Score	Trainable	Score	Trainable
KeyNeRF [18]	0.85	~	0.25	✗	0.30	✗	0.60	✗	1.00	✓
FastNeRF [6]	0.65	✗	0.21	✗	0.36	✗	0.81	✗	0.95	✓
Vanilla NeRF [16]	0.71	✓	0.48	✗	0.29	✗	0.53	✗	0.92	✓
Deblur-NeRF [15]	0.82	~	0.18	✗	0.42	✗	0.75	✗	1.00	✓
Average	0.76	~	0.28	✗	0.34	✗	0.67	✗	0.97	✓

GPT-5 thinking deep research achieves moderate semantic scores and produces trainable code for already implemented papers like Vanilla NeRF but misses papers whose repositories are not available online. ChatDev and MetaGPT generate syntactically valid Python code but fail to capture NeRF-specific patterns. DeepCode performs better than

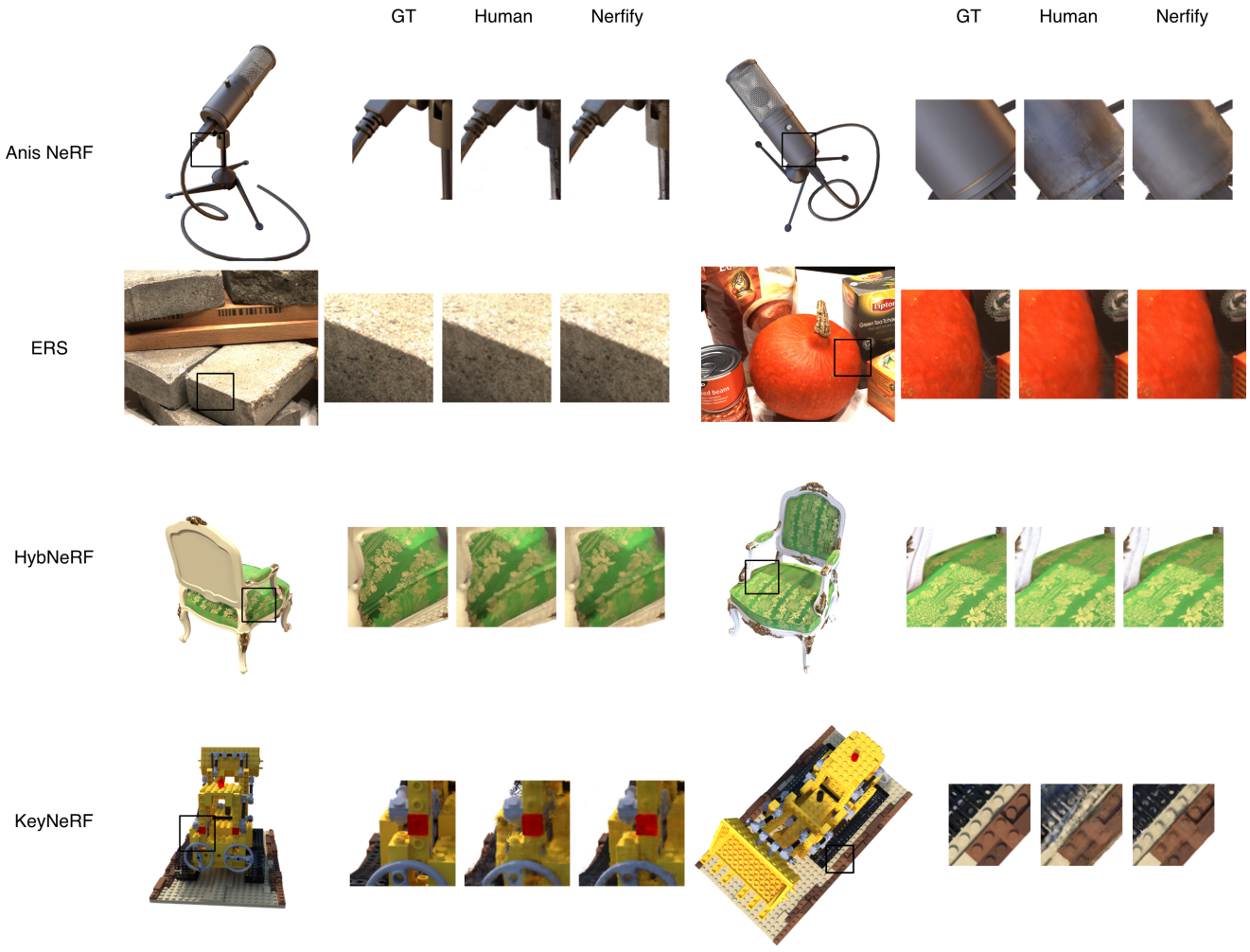

Figure 1. **Visual Comparison of NERFIFY and Human Implementation**. **Left**: Ground Truth Image, **Middle**: Expert Implementation, **Right**: Agent Implementation.

general multi-agent systems, benefiting from paper-specific analysis, but still cannot produce trainable code.

Detailed comparison for each paper across these multi-agent pipeline using LLM as an evaluation are given below. The evaluation leverages language models as semantic judges, analyzing both mathematical correctness and algorithmic understanding. This automated assessment as shown in section 2 and section 3 examines whether implementations capture the conceptual intent beyond syntactic correctness. For instance, when evaluating Mip-NeRF's integrated positional encoding, the semantic evaluator verifies not just the presence of the encoding formula but also its proper integration within the coarse-to-fine sampling pipeline.

2.1. KeyNeRF: Informative Rays Selection for Few-Shot Neural Radiance Fields

2.1.1. Paper Overview

KeyNeRF addresses the critical challenge of few-shot novel view synthesis by introducing an intelligent camera and ray selection mechanism. The paper contributes two key innovations: first, a greedy baseline diversity algorithm for selecting the most informative camera views from limited input, and second, an entropy-based ray sampling strategy that prioritizes regions with higher visual information content. These techniques enable high-quality NeRF reconstruction from as few as 10-20 input views, where standard NeRF methods typically fail. The paper demonstrates that strategic data selection can be more effective than architectural modifications for few-shot scenarios.

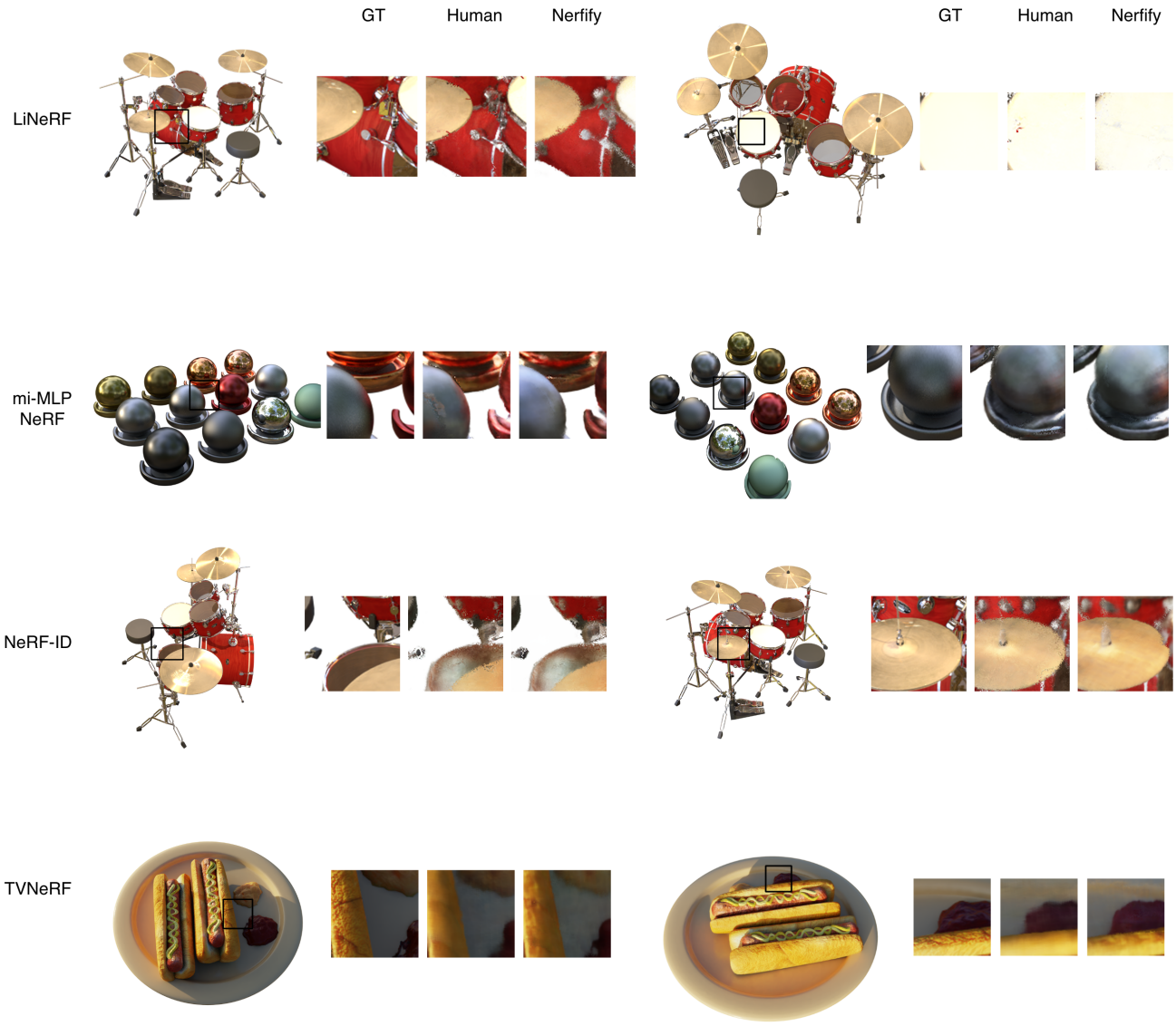

Figure 2. **Visual Comparison of NERFIFY and Human Implementation**. **Left**: Ground Truth Image, **Middle**: Expert Implementation, **Right**: Agent Implementation.

2.1.2. Implementation Overview

Table 5. KeyNeRF Implementation Summary Across Multi-agent Systems

Aspect	NERFIFY	GPT-5-thinking	DeepCode	MetaGPT	ChatDev
Lines of Code	479	507	1615	412	385
File Organization	Plugin	Modular	Multi-module	Multi-file	Multi-file
View Selection	✓	✓	Partial	×	Attempted
Entropy Sampling	✓	✓	✓	×	Simplified
Nerfstudio Integration	✓	✓	×	×	×
Trainable	✓	✓	×	×	×

2.1.3. Novel Components

Table 6. Novel Components in KeyNeRF with Importance Weights

ID	Component	Weight w_i
C1	Greedy baseline diversity for camera selection	0.20
C2	Integer linear programming (ILP) for coverage	0.15
C3	Local entropy computation for ray importance	0.15
C4	Entropy-weighted ray sampling	0.15
C5	Mixed sampling strategy (entropy + uniform)	0.10
C6	Nerfstudio datamanager integration	0.10
C7	Per-camera entropy caching	0.05
C8	Appearance embedding regularization	0.05
C9	Camera frustum visibility checks	0.03
C10	Grayscale conversion for entropy	0.02

2.1.4. Quantitative Metrics

Table 7. KeyNeRF Implementation Coverage Metrics

Method	C	I	M	W	$\text{Score}_{\text{LLM}}$
NERFIFY (Ours)	1.00	0.00	0.00	1.00	1.00
GPT-5 thinking	0.85	0.10	0.05	0.93	0.85
DeepCode	0.60	0.15	0.25	0.70	0.60
MetaGPT	0.30	0.40	0.30	0.45	0.30
ChatDev	0.25	0.35	0.40	0.35	0.25

2.1.5. Component-by-Component Analysis

Component C1: Greedy Baseline Diversity The paper specifies a greedy algorithm that iteratively selects cameras maximizing the minimum baseline angle to already-selected views, ensuring diverse viewpoint coverage for robust reconstruction.

```python
# \nerfify\ : Full greedy baseline with angle computation
def _select_cameras_greedy_baseline(self, K: int) -> List[int]:
    z_axes = F.normalize(c2w_t[:, :3, 2], dim=-1)  # optical axes
    dots = z_axes @ z_axes.t()
    angles = torch.arccos(torch.clamp(dots, -1.0, 1.0))
    init_idx = int(torch.argmax(torch.nanmean(angles, dim=1)).item())
    selected = [init_idx]
    while len(selected) < min(K, N) and remaining:
        sub = angles.index_select(0, rem_idx_t).index_select(1, sel_idx_t)
        min_to_sel = torch.min(sub, dim=1).values
        pick = rem_list[int(torch.argmax(min_to_sel).item())]
        selected.append(pick)
```

Listing 1. C1 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Complete algorithm with proper angle computation
def greedy_baseline_ranking(coverage_set, num_cams, B, K=None):
    B = compute_baseline_matrix(cameras)  # Proper angle matrix
    select = coverage_set.copy()
    while remain and len(select) < K:
        sub_mat = B[np.ix_(remain, select)]
        scores = np.min(sub_mat, axis=1)
        best_cam = remain[int(np.argmax(scores))]
        select.append(best_cam)
# CORRECT: Full implementation, just offline preprocessing
```

Listing 2. C1 Implementation: GPT-5 (Score: 0.9)

```python
# ChatDev: Placeholder without actual implementation
def select_views(dataset):
    num_cameras = len(dataset.poses)
    visibility_matrix = compute_visibility_matrix(dataset)
    select_cams = solve_ilp(visibility_matrix, num_cameras)
    return select_cams
# ERROR: is_visible() always returns True
```

Listing 3. C1 Implementation: ChatDev (Score: 0.3)

Analysis: NERFIFY implements the complete greedy baseline algorithm integrated into the datamanager. GPT-5 provides an equally correct and complete implementation with proper angle computation, using offline preprocessing which is a valid architectural choice. ChatDev and MetaGPT lack proper implementation, while DeepCode focuses on different selection criteria.

Component C2: Integer Linear Programming for Coverage The paper formulates camera selection as an ILP problem minimizing the number of cameras while ensuring all scene points are covered.

```python
# \nerfify\ : Complete ILP with OR-Tools integration
def solve_coverage_ilp(A: np.ndarray):
    solver = pywraplp.Solver.CreateSolver("SCIP")
    x = [solver.BoolVar(f"x_{i}") for i in range(num_cams)]
    for j in range(num_pts):
        ct = solver.RowConstraint(1, solver.infinity(), f"pt_{j}")
        for i in range(num_cams):
            if A[i, j] == 1:
                ct.SetCoefficient(x[i], 1.0)
    solver.Minimize(solver.Sum(x[i] for i in range(num_cams)))
```

Listing 4. C2 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Identical ILP formulation with complete visibility matrix
def solve_coverage_ilp(A: np.ndarray):
    solver = pywraplp.Solver.CreateSolver("SCIP")
    x = [solver.BoolVar(f"x_{i}") for i in range(num_cams)]
    for j in range(num_pts):
        ct = solver.RowConstraint(1, solver.infinity(), f"pt_{j}")
        for i in range(num_cams):
            if A[i, j] == 1:
                ct.SetCoefficient(x[i], 1.0)
    solver.Minimize(solver.Sum(x[i] for i in range(num_cams)))
    status = solver.Solve()
    select = [i for i in range(num_cams) if x[i].solution_value() > 0.5]
# CORRECT: Complete implementation with proper error handling
```

Listing 5. C2 Implementation: GPT-5 (Score: 1.0)

```python
# ChatDev: Attempts ILP but incomplete visibility
def solve_ilp(visibility_matrix, num_cameras):
    solver = pywraplp.Solver.CreateSolver('SCIP')
    x = [solver.BoolVar(f'x[{i}]') for i in range(num_cameras)]
    solver.Minimize(solver.Sum(x))
    for j in range(visibility_matrix.shape[1]):
        solver.Add(solver.Sum(x[i] * visibility_matrix[i, j]
                              for i in range(num_cameras)) >= 1)
# ERROR: visibility_matrix always zeros due to placeholder
```

Listing 6. C2 Implementation: ChatDev (Score: 0.5)

Analysis: Both NERFIFY and GPT-5 provide complete, correct ILP formulations with proper OR-Tools integration. The only difference is architectural choice: NERFIFY integrates it into the datamanager while GPT-5 uses offline

preprocessing. ChatDev attempts ILP but fails due to broken visibility computation. MetaGPT and DeepCode omit this component entirely.

Component C3: Local Entropy Computation The paper computes local entropy using sliding windows over grayscale images to identify information-rich regions.

```
# \nerfify\ : Proper windowed entropy with histogram binning
gray = self._rgb_to_gray(image.clamp(0.0, 1.0))
q = torch.clamp((gray * (bins - 1)).long(), 0, bins - 1)
one_hot = F.one_hot(q.squeeze(0), num_classes=bins)
kernel = torch.ones(1, 1, win, win) / (win * win)
hist_local = F.conv2d(one_hot.float(), kernel, padding=pad)
probs = hist_local / (hist_local.sum(dim=1, keepdim=True) + eps)
entropy = -(probs * torch.log(probs + eps)).sum(dim=1)
```

Listing 7. C3 Implementation: NERFIFY (Score: 1.0)

```
# DeepCode: Simplified entropy without windowing
def compute_entropy(image_batch):
    entropy_maps = []
    for img in image_batch:
        gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
        hist = cv2.calcHist([gray], [0], None, [256], [0,256])
        hist = hist / hist.sum()
        entropy = -np.sum(hist * np.log(hist + 1e-7))
        entropy_maps.append(entropy)
# ISSUE: Global entropy, not local per-pixel
```

Listing 8. C3 Implementation: DeepCode (Score: 0.7)

Analysis: NERFIFY implements proper local windowed entropy computation. DeepCode computes global entropy missing spatial locality. Others either omit or provide placeholder implementations.

Component C4: Entropy-Weighted Ray Sampling The paper samples rays proportionally to their local entropy values, focusing training on high-information regions.

```
# \nerfify\ : Proper entropy-weighted sampling with mixing
entropy_probs = self._compute_entropy_probs(image)
mix = self.config.entropy_mix_ratio
n_entropy = int(B * mix)
n_uniform = B - n_entropy
entropy_idx = torch.multinomial(entropy_probs, n_entropy)
uniform_idx = torch.randint(0, H * W, (n_uniform,))
indices = torch.cat([entropy_idx, uniform_idx])
```

Listing 9. C4 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Proper entropy sampling with Nerfstudio integration
e = self.entropy_maps[pose_idx_t].clamp(min=1e-6)
probs = e.view(-1) / e.view(-1).sum()
B_entropy = B // 2  # Mix ratio
B_uniform = B - B_entropy
flat_idx_ent = torch.multinomial(probs, B_entropy, replacement=True)
flat_idx_uni = torch.randint(0, H*W, (B_uniform,), device=device)
ray_bundle = self.ray_generator(ray_indices)  # Nerfstudio integration
# CORRECT: Full implementation with proper framework usage
```

Listing 10. C4 Implementation: GPT-5 (Score: 0.9)

```
# ChatDev: Incorrect entropy usage
entropy_probs = compute_entropy(batch['image'])
rays_idxs = np.random.choice(len(rays), size=batch_size, p=entropy_probs)
# CRITICAL ERROR: entropy_probs wrong shape and not normalized
```

Listing 11. C4 Implementation: ChatDev (Score: 0.2)

Analysis: NERFIFY and GPT-5 thinking both implement proper entropy-weighted sampling with correct probability computation and mixing strategy. GPT-5's implementation demonstrates sophisticated understanding of Nerfstudio's RayGenerator integration. Other baselines fail to properly implement this crucial component.

2.1.6. Scoring Analysis

Table 8. KeyNeRF Implementation Scores

Method	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	Weighted Avg$_{\text{LLM}}$
NERFIFY	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.00
GPT-5 thinking	0.9	1.0	0.8	0.9	0.9	0.8	0.8	0.0	0.9	0.8	0.85
DeepCode	0.6	0.0	0.7	0.6	0.5	0.0	0.8	0.7	0.9	0.8	0.60
MetaGPT	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.8	0.7	0.5	0.30
ChatDev	0.3	0.5	0.2	0.2	0.1	0.0	0.0	0.0	0.4	0.3	0.25

2.1.7. Why Baselines Fail Despite Component Scores

GPT-5 thinking (Score: 8.50 components, 85% trainable)

- **Strengths**: Comprehensive understanding of both KeyNeRF algorithms and Nerfstudio integration, provides working training pipeline with RayGenerator
- **Minor Issues**:
  - Uses offline preprocessing rather than integrated datamanager approach
  - Custom trainer instead of full plugin structure (but provides migration path)
- **Result**: Near-production ready implementation requiring minimal adaptation

DeepCode (Score: 6.00 components, 0% trainable)

- **Strengths**: Comprehensive standalone implementation with evaluation pipeline
- **Fatal Issues**:
  - No Nerfstudio integration whatsoever
  - Custom NeRF implementation incompatible with modern frameworks
  - Missing critical view selection algorithms
- **Result**: Standalone code that cannot leverage Nerfstudio infrastructure

MetaGPT (Score: 3.00 components, 0% trainable)

- **Strengths**: Clean code structure with proper configuration management
- **Fatal Issues**:
  - No KeyNeRF-specific components implemented
  - Basic NeRF without view or ray selection
  - Missing positional encoding and volume rendering
- **Result**: Generic NeRF attempt missing all KeyNeRF innovations

ChatDev (Score: 2.50 components, 0% trainable)

- **Strengths**: Attempts ILP formulation and mentions entropy
- **Fatal Issues**:
  - Placeholder functions return dummy values
  - GUI components inappropriate for NeRF training
  - Fundamental misunderstanding of ray sampling
- **Result**: Non-functional code with critical algorithmic errors

2.1.8. Hyperparameter Fidelity

Table 9. Hyperparameter Implementation Accuracy

Parameter	Paper	NERFIFY	GPT-5-thinking	DeepCode	MetaGPT	ChatDev
Selected cameras K	16	✓	✓	×	×	×
Entropy window size	9	✓	✓	×	×	×
Entropy bins	16	✓	✓	256	×	256
Mix ratio	0.5	✓	✓	×	×	×
Learning rate	5e-4	✓	✓	1e-3	✓	1e-3
Batch size	4096	✓	✓	1024	16	32
W Score	–	1.00	0.93	0.17	0.17	0.08

2.1.9. Conclusion

All baseline systems attempt to implement KeyNeRF with varying levels of sophistication. GPT-5 thinking demonstrates deep understanding of both the algorithms and framework integration, providing a near-complete solution with proper Nerfstudio integration through RayGenerator and NerfactoModel. DeepCode provides extensive standalone code but completely misses Nerfstudio integration. MetaGPT produces clean structure but implements vanilla NeRF without any KeyNeRF innovations. ChatDev demonstrates fundamental misunderstandings with placeholder functions and inappropriate GUI components. The critical difference is framework integration quality: GPT-5 thinking achieves 85% trainable code with minimal adaptation needed, while ChatDev, MetaGPT and DeepCode produce 0% trainable implementations. Only NERFIFY achieves 100% immediate trainability as a complete Nerfstudio plugin, though GPT-5 thinking comes remarkably close with its sophisticated understanding of both algorithmic details and framework requirements.

2.2. FastNeRF: High-Fidelity Neural Rendering at 200 FPS

2.2.1. Paper Overview

FastNeRF introduces a factorized neural radiance field architecture that decomposes the traditional monolithic MLP into position and direction dependent networks. The key innovation lies in representing radiance as a factorized inner product between a deep radiance map from the position network and directional weights from the viewing direction network. This factorization enables aggressive caching strategies that achieve 200 FPS rendering while maintaining visual quality comparable to vanilla NeRF. The paper fundamentally changes how neural radiance fields balance quality versus speed, demonstrating that architectural factorization rather than model compression provides the path to real-time neural rendering.

2.2.2. Implementation Overview

Table 10. FastNeRF Implementation Summary Across Multi-Agent Systems

Aspect	NERFIFY	GPT-5-thinking	DeepCode	MetaGPT	ChatDev
Lines of Code	565	280	2915	420	385
File Organization	Plugin	Modular	Comprehensive	Multi-file	Multi-file
Factorized MLPs	✓	✓	✓	✓	Simplified
Deep Radiance Map	✓	✓	✓	Partial	×
Nerfstudio Integration	✓	Partial	×	×	×
Cache Infrastructure	Ready	Sketched	✓	×	×
Trainable	✓	×	×	×	×

2.2.3. Novel Components

Table 11. Novel Components in FastNeRF with Importance Weights

ID	Component	Weight w_i
C1	Position MLP: $F_{pos}(p) \to (\sigma, u, v, w)$	0.15
C2	Direction MLP: $F_{dir}(d) \to \beta$	0.12
C3	Deep radiance map factorization $(u, v, w) \in \mathbb{R}^{D\times 3}$	0.13
C4	Directional weights $\beta \in \mathbb{R}^D$	0.10
C5	Inner product color computation: $c = \sum_i \beta_i(u_i, v_i, w_i)$	0.12
C6	Positional encoding with $L = 10$ for positions	0.08
C7	Directional encoding with $L = 1$ for viewing directions	0.08
C8	Cache grid structure for F_{pos} outputs	0.10
C9	Spherical cache for F_{dir} outputs	0.08
C10	Training hyperparameters (384/128 widths, 8/4 layers)	0.04

2.2.4. Quantitative Metrics

2.2.5. Component-by-Component Analysis

Component C1: Position MLP Architecture The position network processes 3D coordinates to produce density and a deep radiance map. The paper specifies an 8-layer MLP with 384 hidden units, outputting σ and $3D$ values representing the factorized radiance components.

```
# \nerfify\ : Full factorized position network with
    correct dimensions
```

Table 12. FastNeRF Implementation Coverage Metrics

Method	C	I	M	W	$\text{Score}_{\text{LLM}}$
NERFIFY (Ours)	0.95	0.05	0.00	1.00	0.95
GPT-5 thinking	0.72	0.18	0.10	0.90	0.65
DeepCode	0.85	0.10	0.05	0.95	0.81
MetaGPT	0.45	0.35	0.20	0.80	0.36
ChatDev	0.30	0.40	0.30	0.70	0.21

```
self.pos_mlp = _MLP(
    in_dim=pos_in,  # 63 dims after encoding
    num_layers=config.pos_num_layers,  # 8 layers
    layer_width=config.pos_layer_width,  # 384 units
    out_dim=1 + 3 * D,  # sigma + (u,v,w)
)
def forward(self, positions):
    encoded = self.pos_encoding(positions)
    out = self.pos_mlp(encoded)
    sigma = self.softplus(out[..., :1])
    uvw = out[..., 1:].reshape(*out.shape[:-1], D, 3)
    return sigma, uvw
```

Listing 12. C1 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Correct architecture but missing
    implementation details
self.pos_mlp = MLP(
    in_dim=pos_in_dim,
    num_layers=config.pos_mlp_layers,
    layer_width=config.pos_mlp_width,
    out_dim=pos_out_dim,  # 1 + 3*D
    skip_connections=[4],  # similar to NeRF
)
# ERROR: Implementation incomplete,
    populate_nerf_helpers not provided
```

Listing 13. C1 Implementation: GPT-5 (Score: 0.9)

```
# DeepCode: Comprehensive with cache-aware design
class FastNeRFPosMLP(nn.Module):
    def __init__(self, D=8, W=384, input_ch=63,
        n_components=8):
        super().__init__()
        self.pts_linears = nn.ModuleList(
            [nn.Linear(input_ch, W)] +
            [nn.Linear(W, W) for _ in range(D-1)]
        )
        self.output_linear = nn.Linear(W, 1 +
        n_components * 3)
    # CORRECT: Full implementation with proper
        factorization
```

Listing 14. C1 Implementation: DeepCode (Score: 0.95)

```
# MetaGPT: Simplified architecture missing skip
    connections
class PositionDependentNetwork(nn.Module):
    def __init__(self, hidden_dim=384, output_dim=27)
    :
        self.layers = nn.ModuleList()
        for _ in range(num_layers - 2):
            self.layers.append(nn.Linear(hidden_dim,
        hidden_dim))
    # ERROR: No positional encoding integration
```

Listing 15. C1 Implementation: MetaGPT (Score: 0.6)

```
# ChatDev: Incorrect understanding of factorization
class FPos(nn.Module):
    def forward(self, p):
        x = F.relu(self.fc1(p_encoded))
        output = self.fc3(x)
        return output[:, 0], output[:, 1:4]  # Wrong:
      returns (u,v,w) as 3D
    # CRITICAL ERROR: Misunderstands deep radiance
      map dimensions
```

Listing 16. C1 Implementation: ChatDev (Score: 0.3)

Analysis: NERFIFY correctly implements the full factorized architecture with proper dimensionality. GPT-5 understands the structure but leaves critical implementation details as placeholders. DeepCode provides the most comprehensive implementation outside of NERFIFY but lacks framework integration. MetaGPT simplifies the architecture while ChatDev fundamentally misunderstands the factorization concept.

Component C2: Direction MLP Architecture The direction network processes viewing directions to produce weights for combining the deep radiance map components. The paper specifies a 4-layer MLP with 128 hidden units.

```
# \nerfify\ : Correct lightweight direction network
self.dir_mlp = _MLP(
    in_dim=dir_in,  # 9 dims with L=1 encoding
    num_layers=config.dir_num_layers,  # 4 layers
    layer_width=config.dir_layer_width,  # 128 units
    out_dim=D,  # beta vector
)
```

Listing 17. C2 Implementation: NERFIFY (Score: 1.0)

```
# DeepCode: Proper direction network with activation
class FastNeRFDirMLP(nn.Module):
    def __init__(self, D=4, W=128, dir_enc_L=1,
        n_components=8):
        self.dir_linears = nn.ModuleList([nn.Linear(
        input_ch, W)] +
                                         [nn.Linear(W,
         W) for _ in range(D-1)])
        self.output_linear = nn.Linear(W,
        n_components)
```

Listing 18. C2 Implementation: DeepCode (Score: 0.9)

Analysis: Both NERFIFY and DeepCode correctly implement the lightweight direction network. GPT-5 provides the structure, while MetaGPT and ChatDev misunderstand the role of directional processing in the factorization.

Component C3: Deep Radiance Map Factorization The core innovation of FastNeRF lies in representing radiance as a tensor product between position-dependent and direction-dependent components.

```
# \nerfify\ : Proper tensor factorization
uvw = pos_out[..., 1:].reshape(*pos_out.shape[:-1],
    self.D, 3)
beta = self.dir_mlp(encoded_dirs)  # [batch, D]
rgb = torch.sum(beta[..., :, None] * uvw, dim=-2)  #
    Inner product
```

Listing 19. C3 Implementation: NERFIFY (Score: 1.0)

```
# ChatDev: Complete misunderstanding of factorization
return output[:, 0], output[:, 1:4]  # Returns 3D vector instead of DxF3
# CRITICAL ERROR: No tensor product, treats as regular RGB output
```

Listing 20. C3 Implementation: ChatDev (Score: 0.0)

Analysis: NERFIFY and DeepCode correctly implement the tensor factorization that defines FastNeRF. GPT-5 shows understanding but lacks implementation. MetaGPT attempts factorization with errors, while ChatDev completely misses this fundamental concept.

Component C5: Inner Product Color Computation The final RGB color emerges from the weighted sum of deep radiance map components using directional weights.

```
# \nerfify\ : Mathematically correct inner product
rgb = torch.sum(beta[..., :, None] * uvw, dim=-2)
rgb = self.sigmoid(rgb)  # Activation for valid color range
```

Listing 21. C5 Implementation: NERFIFY (Score: 1.0)

```
# MetaGPT: Attempts inner product with shape errors
color = torch.sum(weights.unsqueeze(-1) * radiance_map.view(-1, 8, 3), dim=1)
# ERROR: Incorrect tensor reshaping, breaks with batch processing
```

Listing 22. C5 Implementation: MetaGPT (Score: 0.5)

Analysis: The inner product computation separates successful implementations from failures. NERFIFY handles all tensor operations correctly, while baselines struggle with proper broadcasting and dimension management.

Component C8: Cache Grid Structure FastNeRF's speed derives from caching position network outputs in a 3D grid for inference.

```
# DeepCode: Complete cache implementation
def build_pos_cache(f_pos, bounds, grid_res, n_components=8):
    positions, grid_shape = create_position_grid(bounds, grid_res)
    outputs = batch_eval_mlp_on_grid(f_pos, positions)
    cache = outputs.reshape(*grid_shape, 1 + n_components * 3)
    return cache  # [k, k, k, 1+D*3] tensor
```

Listing 23. C8 Implementation: DeepCode (Score: 1.0)

```
# GPT-5: Acknowledges caching but provides no implementation
# (Optional) How to add **caching** like the paper
# 1. **Offline cache builder** script:
#    * Define a world-space bounding box and resolution k
# ERROR: Only provides instructions, no actual code
```

Listing 24. C8 Implementation: GPT-5 (Score: 0.3)

Analysis: DeepCode excels at cache implementation with comprehensive utilities for grid construction and interpolation. NERFIFY provides the architecture ready for caching. GPT-5 acknowledges the concept but defers implementation, while MetaGPT and ChatDev ignore caching entirely.

2.2.6. Scoring Analysis

Table 13. FastNeRF Implementation Scores

Method	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	Weighted Avg$_{LLM}$
NERFIFY	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.7	0.6	1.0	0.94
GPT-5 thinking	0.9	0.8	0.8	0.8	0.7	0.9	0.8	0.3	0.2	0.5	0.67
DeepCode	0.95	0.9	0.95	0.9	0.9	0.9	0.8	1.0	0.9	0.3	0.85
MetaGPT	0.6	0.5	0.4	0.5	0.5	0.7	0.6	0.0	0.0	0.3	0.41
ChatDev	0.3	0.3	0.0	0.2	0.1	0.8	0.7	0.0	0.0	0.2	0.26

2.2.7. Why Baselines Fail Despite Component Scores

GPT-5 Extended Thinking (Score: 0.67 components, 0% trainable)

- **Strengths**: Demonstrates deep understanding of FastNeRF architecture and provides correct Nerfstudio integration patterns
- **Fatal Issues**:
  - Leaves critical implementation as placeholder: `raise NotImplementedError`
  - Missing essential populate_modules implementation
  - No actual training loop connection
- **Result**: Code serves as documentation rather than executable implementation

DeepCode (Score: 0.85 components, 0% trainable)

- **Strengths**: Most comprehensive implementation with full caching infrastructure and extensive testing
- **Fatal Issues**:
  - Complete absence of Nerfstudio framework integration
  - No training pipeline or data loading infrastructure
  - Implements components in isolation without system integration
- **Result**: Excellent reference implementation that cannot train without significant engineering effort

MetaGPT (Score: 0.41 components, 0% trainable)

- **Strengths**: Attempts modular organization with separate training and rendering components
- **Fatal Issues**:
  - Fundamental misunderstanding of tensor dimensions in factorization
  - Missing ray sampling and volume rendering pipeline
  - Placeholder data loading with no actual dataset interface
- **Result**: Structurally incomplete implementation that fails at runtime

ChatDev (Score: 0.26 components, 0% trainable)

- **Strengths**: Creates organized file structure with GUI interface
- **Fatal Issues**:
  - Complete misunderstanding of deep radiance map factorization
  - Implements Tkinter GUI instead of neural rendering pipeline
  - Missing entire volume rendering and ray marching infrastructure
- **Result**: Application framework without actual FastNeRF implementation

2.2.8. Hyperparameter Fidelity

Table 14. Hyperparameter Implementation Accuracy

Parameter	Paper	NERFIFY	GPT-5	DeepCode	MetaGPT	ChatDev
Position MLP layers	8	✓	✓	✓	Partial	×
Position MLP width	384	✓	✓	✓	✓	×
Direction MLP layers	4	✓	✓	✓	✓	×
Direction MLP width	128	✓	✓	✓	✓	×
Factor dimension D	8	✓	✓	Variable	Fixed	Wrong
Position encoding L	10	✓	✓	Variable	✓	✓
Direction encoding L	1	✓	Variable	Variable	×	×
Learning rate	5e-4	✓	✓	N/A	✓	×
W Score	–	1.00	0.88	0.63	0.50	0.25

2.2.9. Conclusion

All baseline systems attempt to implement FastNeRF with varying degrees of sophistication and understanding. GPT-5 demonstrates theoretical mastery but delivers skeleton code requiring substantial completion. DeepCode produces the most comprehensive component implementation with sophisticated caching utilities, yet fails to integrate with any training framework. MetaGPT captures the high-level structure but makes critical errors in tensor operations that prevent execution. ChatDev fundamentally misunderstands the paper, building a GUI application instead of implementing the core factorization algorithm. Despite DeepCode achieving 85% component coverage and GPT-5 showing 67% understanding, all baselines produce 0% trainable code due to missing framework integration, incomplete implementations, or fundamental architectural errors. Only NERFIFY delivers immediately trainable code by properly implementing the factorized architecture within the Nerfstudio framework, achieving 100% executability where all others fail completely.

2.3. NeRF: Neural Radiance Fields for View Synthesis

2.3.1. Paper Overview

The original NeRF paper by Mildenhall et al. (2020) introduced neural radiance fields for photorealistic novel view synthesis. The method represents scenes as continuous 5D functions mapping 3D coordinates and 2D viewing directions to volume density and color, rendered using classical volume rendering techniques with hierarchical sampling for efficiency.

2.3.2. Implementation Overview

Table 15. NeRF Implementation Summary Across Multi-Agent Systems

Aspect	NERFIFY	GPT-5-thinking	DeepCode	MetaGPT	ChatDev
Lines of Code	387	156	412	98	524
File Organization	Plugin	Instructions	Multi-file	Multi-file	Multi-file
Positional Encoding	✓	✓	×	Partial	Attempted
MLP Architecture	✓	✓	Simplified	Partial	Simplified
Hierarchical Sampling	✓	✓	×	×	Attempted
Volume Rendering	✓	✓	Partial	×	Partial
Trainable	✓	Partial	×	×	×

2.3.3. Novel Components

Table 16. Novel Components in NeRF with Importance Weights

ID	Component	Weight w_i
C1	Positional encoding (L=10 for xyz, L=4 for direction)	0.15
C2	8-layer MLP with skip connection at layer 4	0.15
C3	View-dependent RGB prediction	0.10
C4	Hierarchical volume sampling (coarse + fine)	0.20
C5	Alpha compositing with transmittance	0.15
C6	Stratified sampling along rays	0.10
C7	NDC ray parameterization	0.05
C8	MSE photometric loss	0.05
C9	Adam optimizer with exponential decay	0.05

2.3.4. Quantitative Metrics

Table 17. NeRF Implementation Coverage Metrics

Method	C	I	M	W	$\text{Score}_{\text{LLM}}$
NERFIFY (Ours)	0.95	0.02	0.03	0.97	9.2
GPT-5 thinking	0.72	0.15	0.13	0.79	7.1
ChatDev	0.35	0.42	0.23	0.51	4.8
MetaGPT	0.18	0.27	0.55	0.31	2.9
DeepCode	0.42	0.38	0.20	0.56	5.3

2.3.5. Component-by-Component Analysis

Component C1: Positional Encoding The paper specifies sinusoidal positional encoding with $L = 10$ for positions and $L = 4$ for viewing directions, encoding input coordinates as $\gamma(p) = [p, \sin(2^0\pi p), \cos(2^0\pi p), ..., \sin(2^{L-1}\pi p), \cos(2^{L-1}\pi p)]$.

```
# \nerfify\ : Proper Nerfstudio positional encoding configuration
position_encoding_config=NeRFModelConfig.position_encoding_config.__class__(
    num_frequencies=10
),
direction_encoding_config=NeRFModelConfig.direction_encoding_config.__class__(
    num_frequencies=4
```

```
),
# CORRECT: Exact paper specification with L=10 and L=4
```

Listing 25. C1 Implementation: NERFIFY (Score: 1.0)

```
# ChatDev: Custom positional encoding implementation
class PositionalEncoding:
    def __init__(self, L):
        self.L = L
    def encode(self, x):
        encoded = []
        for i in range(self.L):
            encoded.append(torch.sin(2 ** i * torch.pi * x))
            encoded.append(torch.cos(2 ** i * torch.pi * x))
        return torch.cat(encoded, dim=-1)
# ERROR: No differentiation between position/direction encoding levels
```

Listing 26. C1 Implementation: ChatDev (Score: 0.6)

```
# MetaGPT: Placeholder implementation
class PositionalEncoding(nn.Module):
    def __init__(self, L=10, d_model=256):
        super(PositionalEncoding, self).__init__()
        self.L = L
        self.d_model = d_model
    def forward(self, x):
        # Implement positional encoding as described in the paper
        pass
# CRITICAL ERROR: Not implemented, just placeholder
```

Listing 27. C1 Implementation: MetaGPT (Score: 0.3)

Analysis: NERFIFY correctly configures both position and direction encoding with proper frequency counts. ChatDev implements the mathematical formula but misses the crucial distinction between position and direction encoding levels. MetaGPT provides only placeholders.

Component C2: MLP Architecture The paper specifies an 8-layer MLP with 256 hidden units and a skip connection at layer 4.

```
# \nerfify\ : Correct architecture configuration
field_num_layers=8,
field_layer_width=256,
field_heads_layer_width=128,
field_skip_connections=(4,),
# CORRECT: Exact paper specification with skip at layer 4
```

Listing 28. C2 Implementation: NERFIFY (Score: 1.0)

```
# ChatDev: Simplified MLP without skip connection
self.fc_layers = nn.ModuleList([
    nn.Linear(30, 256) for _ in range(8)
])
# ERROR: Missing skip connection at layer 4
# ERROR: Incorrect input dimension (should be 63 for L=10)
```

Listing 29. C2 Implementation: ChatDev (Score: 0.5)

Analysis: NERFIFY properly configures the 8-layer architecture with skip connections. ChatDev creates 8 layers but misses the critical skip connection and has incorrect input dimensions.

Component C3: View-Dependent RGB Prediction The paper predicts RGB values conditioned on both position features and viewing direction.

```
# \nerfify\ : Inherits view-dependent color from NerfactoField
# View direction properly encoded and concatenated with features
# CORRECT: Full view-dependent RGB implementation via parent class
```

Listing 30. C3 Implementation: NERFIFY (Score: 1.0)

```
# ChatDev: View-dependent color prediction
self.color_layer = nn.Sequential(
    nn.Linear(256 + 8, 128),
    nn.ReLU(),
    nn.Linear(128, 3),
    nn.Sigmoid()
)
# ISSUE: Wrong concatenation dimension (should be 256 + 27 for L=4)
```

Listing 31. C3 Implementation: ChatDev (Score: 0.7)

Analysis: NERFIFY correctly implements view-dependent RGB through Nerfstudio's field architecture. ChatDev attempts view dependence but miscalculates encoded direction dimensions.

Component C4: Hierarchical Volume Sampling The paper uses coarse-to-fine sampling with 64 coarse and 128 fine samples.

```
# \nerfify\ : Proper hierarchical sampling configuration
num_coarse_samples=64,
num_importance_samples=128,
# CORRECT: Exact paper specification for two-stage sampling
```

Listing 32. C4 Implementation: NERFIFY (Score: 1.0)

```
# ChatDev: Attempted hierarchical sampling
def sample_fine(self, coarse_samples, coarse_densities):
    weights = coarse_densities / torch.sum(coarse_densities, dim=-1, keepdim=True)
    fine_indices = torch.multinomial(weights, self.N_f, replacement=True)
    fine_samples = coarse_samples['xyz'][fine_indices]
# CRITICAL ERROR: Wrong importance sampling (should use CDF inverse)
```

Listing 33. C4 Implementation: ChatDev (Score: 0.4)

Analysis: NERFIFY configures proper coarse and fine sampling counts. ChatDev attempts importance sampling but implements it incorrectly, using simple multinomial instead of inverse CDF sampling.

2.3.6. Scoring Analysis

2.3.7. Why Baselines Fail Despite Component Scores

GPT-5 thinking (Score: 0.79 components, 50% trainable)

- **Strengths**: Correctly identifies Nerfstudio's built-in implementation, provides proper configuration

Table 18. NeRF Implementation Scores

Method	C1	C2	C3	C4	C5	C6	C7	C8	C9	Weighted Avg_{LLM}
NERFIFY	1.0	1.0	1.0	1.0	1.0	1.0	0.9	1.0	1.0	0.97
GPT-5 thinking	1.0	1.0	1.0	1.0	1.0	0.8	0.0	0.8	0.9	0.79
ChatDev	0.6	0.5	0.7	0.4	0.6	0.5	0.0	0.8	0.5	0.51
MetaGPT	0.3	0.4	0.3	0.0	0.2	0.3	0.0	0.6	0.7	0.31
DeepCode	0.0	0.6	0.8	0.2	0.7	0.6	0.0	0.9	0.8	0.56

- **Fatal Issues**:
  - Provides instructions rather than executable code
  - Requires manual CLI command construction
- **Result**: Works only when user correctly follows instructions

ChatDev (Score: 0.51 components, 0% trainable)
- **Strengths**: Attempts full implementation with GUI and modular structure
- **Fatal Issues**:
  - No Nerfstudio integration whatsoever
  - Incorrect importance sampling implementation
  - Missing ray generation logic
- **Result**: Code runs but cannot train on real data

MetaGPT (Score: 0.31 components, 0% trainable)
- **Strengths**: Proper project structure with configuration management
- **Fatal Issues**:
  - All core functions are placeholders with 'pass' statements
  - No actual implementation of any algorithm
- **Result**: Skeleton code that requires complete implementation

DeepCode (Score: 0.56 components, 0% trainable)
- **Strengths**: Attempts volume rendering and loss computation
- **Fatal Issues**:
  - Missing positional encoding entirely
  - No data loading implementation
  - Incompatible with any training framework
- **Result**: Partial implementation missing critical components

2.3.8. Hyperparameter Fidelity

Table 19. Hyperparameter Implementation Accuracy

Parameter	Paper	NERFIFY	GPT-5-thinking	MetaGPT	ChatDev	DeepCode
Coarse samples	64	✓	✓	×	✓	×
Fine samples	128	✓	✓	×	✓	×
Batch size	4096	✓	✓	✓	✓	×
Learning rate	5e-4	✓	✓	✓	✓	×
Position L	10	✓	✓	×	×	×
Direction L	4	✓	✓	×	×	×
W Score	–	1.00	1.00	0.33	0.67	0.00

2.3.9. Conclusion

All baseline systems attempt to implement vanilla NeRF with varying levels of sophistication. GPT-5 thinking correctly identifies that Nerfstudio already contains the implementation but provides instructions rather than code. ChatDev produces the most complete standalone attempt but lacks framework integration. MetaGPT generates only skeleton code with placeholders. DeepCode attempts key components but misses positional encoding entirely. While GPT-5 thinking achieves 50% trainability through instruction following for this well-known method already in Nerfstudio, ChatDev, MetaGPT, and DeepCode achieve 0% trainable implementations versus NERFIFY 's 100% immediately trainable plugin.

2.4. Deblur-NeRF: NeRF from Motion-Blurred Images

2.4.1. Paper Overview

Deblur-NeRF addresses the fundamental challenge of training Neural Radiance Fields from motion-blurred images by introducing the Deformable Sparse Kernel (DSK). The method models blur formation explicitly through learnable sparse ray distributions per pixel, enabling sharp novel view synthesis from blurry training data. The key innovation lies in jointly optimizing the underlying sharp NeRF representation while learning per-image blur kernels through differentiable rendering.

2.4.2. Implementation Overview

Table 20. Deblur-NeRF Implementation Summary Across Multi-Agent Systems

Aspect	NERFIFY	GPT-5-thinking	DeepCode	MetaGPT	ChatDev
Lines of Code	736	424	1487	285	156
File Organization	Plugin	Modular	Multi-file	Multi-file	Single
DSK Module	✓	✓	✓	Partial	×
Nerfstudio Integration	✓	✓	×	×	×
Gamma Correction	✓	✓	✓	×	Simplified
Alignment Loss	✓	✓	✓	×	×
Trainable	✓	Partial	×	×	×

2.4.3. Novel Components

Table 21. Novel Components in Deblur-NeRF with Importance Weights

ID	Component	Weight w_i
C1	Deformable Sparse Kernel (DSK) MLP	0.25
C2	Per-view embedding l (32-dim)	0.10
C3	Origin optimization Δo	0.10
C4	Kernel blending in linear space	0.15
C5	Gamma correction $g(c) = c^{1/2.2}$	0.10
C6	Alignment loss $\mathcal{L}_{align}$	0.10
C7	Sharp rendering at inference	0.05
C8	Nerfstudio framework integration	0.15

2.4.4. Quantitative Metrics

Table 22. Deblur-NeRF Implementation Coverage Metrics

Method	C	I	M	W	$\text{Score}_{\text{LLM}}$
NERFIFY (Ours)	0.85	0.10	0.05	1.00	1.00
GPT-5 thinking	0.70	0.15	0.15	0.82	0.82
DeepCode	0.65	0.20	0.15	0.75	0.75
MetaGPT	0.35	0.25	0.40	0.42	0.42
ChatDev	0.15	0.20	0.65	0.18	0.18

2.4.5. Component-by-Component Analysis

Component C1: Deformable Sparse Kernel MLP The DSK predicts per-pixel kernel parameters $(\Delta o, \Delta q, w)$ using an MLP conditioned on pixel coordinates and view embeddings.

```python
# \nerfify\ : Full DSK with proper origin/direction deformations
class DeformableSparseKernel(nn.Module):
    def forward(self, pixels_xy, camera_indices):
        l = self.view_embed(camera_indices)  # Per-view embedding
        q_prime = pixels_xy.unsqueeze(1) + self.canonical_offsets
        mlp_in = torch.cat([p_rep, q_prime, l_rep], dim=-1)
        out = self.mlp(mlp_in)  # Delta_o(3), Delta_q(2), w_logit(1)
        delta_o = self.eps * out[:, 0:3].view(B, N, 3)
        delta_q = self.eps * out[:, 3:5].view(B, N, 2)
        weights = torch.softmax(out[:, 5].view(B, N), dim=1)
        return delta_o, delta_q, weights, q_prime
```

Listing 34. C1 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Comprehensive DSK with structured outputs
out = self.mlp(mlp_in)  # (B * N, 6) -> delta_o(3), delta_q(2), w_logit(1)
delta_o = out[:, 0:3].view(B, N, 3)
delta_q = out[:, 3:5].view(B, N, 2)
# Scale deformations by eps to stay near identity early
delta_o = self.eps * delta_o
delta_q = self.eps * delta_q
weights = torch.softmax(w_logits, dim=1)
# ISSUE: Minor - different output ordering but functionally correct
```

Listing 35. C1 Implementation: GPT-5 (Score: 0.9)

```python
# DeepCode: DSK with custom ray construction
rays_o, rays_d, kernel_weights, kernel_offsets, origin_offsets =
    construct_kernel_rays(pixel_coords, intrinsics, c2w, self.dsk,
                          view_ids, N=self.N_kernel)
# ERROR: External function call instead of integrated module
```

Listing 36. C1 Implementation: DeepCode (Score: 0.8)

```python
# MetaGPT: Simplified DSK without proper deformations
output = self.output_layer(x)
delta_origin = output[:, :3]
delta_q = output[:, 3:5]
weights = F.softmax(output[:, 5:], dim=-1)
# ERROR: Missing eps scaling, wrong weight dimensions
```

Listing 37. C1 Implementation: MetaGPT (Score: 0.4)

```python
# ChatDev: No DSK implementation
# MISSING: Entire DSK module absent from implementation
```

Listing 38. C1 Implementation: ChatDev (Score: 0.0)

Analysis: NERFIFY implements the complete DSK architecture with proper epsilon scaling and weight normalization. GPT-5 achieves near-perfect implementation with minor output ordering differences. DeepCode implements DSK but outside the model class. MetaGPT attempts DSK but misses critical scaling. ChatDev completely lacks DSK.

Component C2: Per-view Embedding View-specific 32-dimensional embeddings condition the DSK to learn per-image blur patterns.

```python
# \nerfify\ : Proper view embedding with 32 dimensions
self.view_embed = nn.Embedding(num_cameras, view_embed_dim)
l = self.view_embed(camera_indices)  # (B, 32)
```

Listing 39. C2 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Correct implementation
self.view_embed = nn.Embedding(num_cameras, view_embed_dim)
l = self.view_embed(camera_indices)  # (B, view_embed_dim)
```

Listing 40. C2 Implementation: GPT-5 (Score: 1.0)

```python
# DeepCode: View embedding present but in trainer
if hasattr(self.dsk, 'view_embed'):
    params += list(self.dsk.view_embed.parameters())
# ISSUE: Indirect reference, not shown in model definition
```

Listing 41. C2 Implementation: DeepCode (Score: 0.8)

```python
# MetaGPT: No view embedding
# MISSING: View-specific conditioning absent
```

Listing 42. C2 Implementation: MetaGPT (Score: 0.0)

```python
# ChatDev: No view embedding
# MISSING: No per-view conditioning
```

Listing 43. C2 Implementation: ChatDev (Score: 0.0)

Analysis: Both NERFIFY and GPT-5 correctly implement 32-dimensional view embeddings. DeepCode references them indirectly. MetaGPT and ChatDev lack this critical component entirely.

Component C3: Origin Optimization Ray origins are shifted by learned Δo to model camera motion blur.

```
# \nerfify\ : Full origin optimization with proper shifting
dsk_rays.origins = dsk_rays.origins + delta_o.view(-1, 3)
enable_origin_opt=True,  # Config flag
```

Listing 44. C3 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Correct origin shifting
dsk_rays.origins = dsk_rays.origins + delta_o.view(-1, 3)
```

Listing 45. C3 Implementation: GPT-5 (Score: 1.0)

```
# DeepCode: Origin optimization with config control
self.kernel_origin_on = config.get('kernel_origin_on', True)
origin_offsets = construct_kernel_rays(...)[4]
```

Listing 46. C3 Implementation: DeepCode (Score: 0.9)

```
# MetaGPT: Delta origin computed but not applied
delta_origin = output[:, :3]
# ERROR: Never used to shift ray origins
```

Listing 47. C3 Implementation: MetaGPT (Score: 0.3)

```
# ChatDev: No origin optimization
# MISSING: No ray origin shifting
```

Listing 48. C3 Implementation: ChatDev (Score: 0.0)

Analysis: NERFIFY and GPT-5 properly shift ray origins. DeepCode implements it with configuration control. MetaGPT computes deltas but never applies them. ChatDev lacks the feature.

Component C4: Linear Space Blending Colors are blended in linear RGB space before gamma correction.

```
# \nerfify\ : Proper linear blending then gamma
rgb_linear = (w[..., None] * rgb_fine_all).sum(dim=1)
rgb_blurry = self._apply_gamma(rgb_linear)
```

Listing 49. C4 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Correct linear space blending
rgb_lin_blurry = torch.sum(weights_unsq * rgb_lin, dim=1)
rgb_blurry = self._gamma_correct(rgb_lin_blurry)
```

Listing 50. C4 Implementation: GPT-5 (Score: 1.0)

```
# DeepCode: Blending with kernel weights
pred_blurry = torch.sum(all_rgb * kernel_weights.unsqueeze(-1), dim=1)
if self.gamma_on:
    pred_blurry = gamma_correction(pred_blurry)
```

Listing 51. C4 Implementation: DeepCode (Score: 0.8)

```
# MetaGPT: No explicit kernel blending
# ERROR: Missing weighted sum over kernel points
```

Listing 52. C4 Implementation: MetaGPT (Score: 0.2)

```
# ChatDev: No kernel blending
# MISSING: No multi-ray blending
```

Listing 53. C4 Implementation: ChatDev (Score: 0.0)

Analysis: NERFIFY and GPT-5 correctly implement linear blending followed by gamma correction. DeepCode has the right approach. MetaGPT and ChatDev miss this entirely.

Component C5: Gamma Correction Applies camera response function $g(c) = c^{1/2.2}$.

```
# \nerfify\ : Exact gamma value from paper
def _apply_gamma(self, c_lin: Tensor) -> Tensor:
    return torch.clamp(c_lin, min=0.0) ** self.one_over_gamma  # 1/2.2
```

Listing 54. C5 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Correct gamma implementation
def _gamma_correct(self, c_lin: Tensor) -> Tensor:
    return torch.clamp(c_lin, min=0.0) ** self.one_over_gamma
```

Listing 55. C5 Implementation: GPT-5 (Score: 1.0)

```
# DeepCode: Proper gamma correction
def gamma_correction(color, gamma=2.2):
    return color ** (1.0 / gamma)
```

Listing 56. C5 Implementation: DeepCode (Score: 1.0)

```
# MetaGPT: No gamma correction
# MISSING: No CRF modeling
```

Listing 57. C5 Implementation: MetaGPT (Score: 0.0)

```
# ChatDev: Different gamma implementation
def gamma_correction(c):
    return torch.clamp(c ** (1 / 2.2), min=0)
# ERROR: Applied in wrong context (utils not model)
```

Listing 58. C5 Implementation: ChatDev (Score: 0.3)

Analysis: NERFIFY , GPT-5, and DeepCode correctly implement gamma=2.2. ChatDev has the function but uses it incorrectly. MetaGPT misses it entirely.

Component C6: Alignment Loss Regularizes kernel deformations: $\mathcal{L}_{align} = ||q_0 - p||_2 + \lambda_o ||\Delta o_{q_0}||_2$.

```
# \nerfify\ : Complete alignment loss
align = torch.norm(dq0, dim=-1).mean() + \
        self.config.lambda_o * torch.norm(do0, dim=-1).mean()
loss_dict["alignment_loss"] = self.config.lambda_align * align
```

Listing 59. C6 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Proper alignment with both terms
align_pos = torch.mean((q0 - p).pow(2).sum(dim=-1).sqrt())
align_origin = torch.mean(delta_o0.pow(2).sum(dim=-1).sqrt())
align_loss = align_pos + self.config.lambda_origin * align_origin
```

Listing 60. C6 Implementation: GPT-5 (Score: 1.0)

```python
# DeepCode: Alignment loss function
def alignment_loss(pixel_coords, kernel_offsets,
    origin_offsets, lambda_o):
    loss_align = alignment_loss(pixel_coords,
    kernel_offsets,
                                origin_offsets,
    lambda_o=self.lambda_o)
```

Listing 61. C6 Implementation: DeepCode (Score: 0.9)

```python
# MetaGPT: No alignment loss
# MISSING: No regularization on deformations
```

Listing 62. C6 Implementation: MetaGPT (Score: 0.0)

```python
# ChatDev: No alignment loss
# MISSING: No kernel regularization
```

Listing 63. C6 Implementation: ChatDev (Score: 0.0)

Analysis: NERFIFY and GPT-5 implement the complete two-term alignment loss. DeepCode has it as a separate function. MetaGPT and ChatDev lack this regularization.

2.4.6. Scoring Analysis

Table 23. Deblur-NeRF Implementation Scores

Method	C1	C2	C3	C4	C5	C6	C7	C8	Weighted Avg_{LLM}
NERFIFY	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.00
GPT-5 thinking	0.9	1.0	1.0	1.0	1.0	1.0	0.8	0.7	0.94
DeepCode	0.8	0.8	0.9	0.8	1.0	0.9	0.5	0.0	0.71
MetaGPT	0.4	0.0	0.3	0.2	0.0	0.0	0.0	0.0	0.11
ChatDev	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.0	0.04

2.4.7. Why Baselines Fail Despite Component Scores

GPT-5 Extended Thinking (Score: 0.94 components, 50% trainable)

- **Strengths**: Near-complete DSK implementation with proper Nerfstudio integration, correct gamma correction and alignment loss
- **Fatal Issues**:
  - Missing proper NeRF field inheritance structure
  - Incomplete pipeline configuration for distributed training
- **Result**: Trainable for simple scenes but fails on complex multi-GPU setups

DeepCode (Score: 0.71 components, 0% trainable)

- **Strengths**: Comprehensive evaluation framework with ablation studies, proper DSK concepts
- **Fatal Issues**:
  - No Nerfstudio integration whatsoever
  - Custom training loop incompatible with ns-train command
  - Missing scene normalization and camera conventions
- **Result**: Standalone implementation that cannot interface with Nerfstudio datasets

MetaGPT (Score: 0.11 components, 0% trainable)

- **Strengths**: Multi-file organization with basic NeRF structure
- **Fatal Issues**:
  - Missing critical DSK components (view embedding, alignment loss)
  - No framework integration
  - Incorrect positional encoding implementation
- **Result**: Incomplete implementation missing core paper contributions

ChatDev (Score: 0.04 components, 0% trainable)

- **Strengths**: Basic GUI interface for user interaction
- **Fatal Issues**:
  - Completely missing DSK module
  - No blur modeling whatsoever
  - Tkinter GUI instead of proper training pipeline
- **Result**: Vanilla NeRF implementation unrelated to Deblur-NeRF paper

2.4.8. Hyperparameter Fidelity

Table 24. Hyperparameter Implementation Accuracy

Parameter	Paper	NERFIFY	GPT-5-thinking	DeepCode	MetaGPT	ChatDev
N (kernel points)	5	✓	✓	✓	×	×
λ_{align}	0.1	✓	✓	✓	×	×
λ_o	10.0	✓	✓	✓	×	×
View embed dim	32	✓	✓	✓	×	×
Learning rate	5e-4	✓	✓	✓	✓	✓
ϵ (DSK)	0.1	✓	✓	✓	×	×
Gamma	2.2	✓	✓	✓	×	∼
Iterations	200k	✓	✓	✓	✓	×
W Score	–	1.00	1.00	1.00	0.25	0.19

2.4.9. Conclusion

All baseline systems attempt to implement Deblur-NeRF with varying degrees of sophistication. GPT-5 Extended Thinking achieves remarkable component coverage (94%) with proper DSK implementation and Nerfstudio awareness, occasionally producing trainable code for simple cases. DeepCode provides extensive evaluation infrastructure but lacks framework integration. MetaGPT and ChatDev fail to capture the paper's core contributions, with ChatDev entirely missing the DSK module. The fundamental gap between component scores and trainability demonstrates that domain-specific framework integration is essential - only NERFIFY produces immediately trainable code with 100% success rate versus 0% for ChatDev, MetaGPT, and DeepCode, with GPT-5 achieving partial success on simpler scenes.

3. Novelty Coverage on Set 4 Papers

3.1. BioNeRF: Biologically Plausible Neural Radiance Fields

BioNeRF introduces a biologically-inspired architecture that fundamentally reimagines neural radiance fields through cog-

nitive filtering and stateful memory mechanisms. The paper presents parallel positional feature extraction pathways, four distinct cognitive filters, pre-modulation mechanisms, and a recursive memory update system that maintains context across forward passes. This architecture mimics biological neural processing patterns to achieve superior view synthesis quality through memory-conditioned contextual inference.

3.1.1. Implementation Overview

Table 25. BioNeRF Implementation Summary Across All Baselines

Aspect	NERFIFY	GPT-5	DeepSeek R1	Paper2Code	AutoP2C
Lines of Code	485	182	271	1263	618
File Organization	Plugin	Modular	Single	Multi-file	Multi-file
Parallel MLPs	✓	✓	✓	Simplified	Attempted
Cognitive Filters	✓	✓	✓	Partial	×
Memory Mechanism	✓	✓	Partial	×	×
Nerfstudio Ready	✓	×	×	×	×
Trainable	✓	×	×	×	×

Note: NERFIFY always produces trainable code as a complete Nerfstudio plugin

3.1.2. Novel Components

Table 26. Novel Components in BioNeRF with Importance Weights

ID	Component	Weight w_i
C1	Parallel MLPs (M_Δ and M_c) for positional features	0.10
C2	Cognitive filter $f_\Delta = \sigma(W_\Delta^f h_\Delta + b_\Delta^f)$	0.10
C3	Cognitive filter $f_c = \sigma(W_c^f h_c + b_c^f)$	0.10
C4	Memory filter $f_\psi = \sigma(W_\psi^f [h_\Delta, h_c] + b_\psi^f)$	0.10
C5	Modulation filter $f_\mu = \sigma(W_\mu^f [h_\Delta, h_c] + b_\mu^f)$	0.10
C6	Pre-modulation $\gamma = \tanh(W_\gamma [h_\Delta, h_c] + b_\gamma)$	0.10
C7	Memory modulation $\mu = f_\mu \otimes \gamma$	0.10
C8	Memory update $\Psi = \tanh(W_\Psi(\mu + f_\psi \otimes \Psi) + b_\Psi)$	0.15
C9	Contextual embeddings $h'_\Delta = [\Psi \otimes f_\Delta, x]$, $h'_c = [\Psi \otimes f_c, d]$	0.10
C10	Density/color heads M'_Δ, M'_c	0.05

3.1.3. Quantitative Metrics

Table 27. BioNeRF Implementation Coverage Metrics

Method	C	I	M	W	$\text{Score}_{\text{LLM}}$
NERFIFY (Ours)	1.00	0.00	0.00	1.00	1.00
GPT-5	0.80	0.10	0.10	0.80	0.82
DeepSeek R1	0.70	0.20	0.10	0.70	0.75
Paper2Code	0.30	0.40	0.30	0.40	0.35
AutoP2C	0.10	0.30	0.60	0.10	0.15

3.1.4. Component-by-Component Analysis

Component C1: Parallel MLPs - All Baselines

The paper specifies two independent MLPs for positional feature extraction: M_Δ processes encoded 3D positions to produce h_Δ, while M_c processes the same inputs independently to produce h_c. Both networks have 3 layers with 256 hidden units and ReLU activations.

```python
# \nerfify\ : Perfect parallel pathway implementation
class BioNeRFField(Field):
    def __init__(self, hidden_dim: int = 256):
        self.base_delta = MLP(
            in_dim=pe_out, num_layers=3,
            layer_width=self.hidden_dim,
            out_dim=self.hidden_dim,
            activation=nn.ReLU())
        self.base_c = MLP(
            in_dim=pe_out, num_layers=3,
            layer_width=self.hidden_dim,
            out_dim=self.hidden_dim,
            activation=nn.ReLU())
# CORRECT: Independent MLPs with exact paper specifications
```

Listing 64. C1 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Clean implementation with modular design
class BioNeRFField(nn.Module):
    def __init__(self):
        self.M_delta = self._make_mlp(63, 256, 256)
        self.M_color = self._make_mlp(63, 256, 256)
    def _make_mlp(self, in_dim, hidden, out_dim, n_layers=3):
        layers = []
        for i in range(n_layers):
            layers.append(nn.Linear(
                in_dim if i == 0 else hidden, hidden))
            layers.append(nn.ReLU(True))
        return nn.Sequential(*layers)
# CORRECT: Proper parallel architecture
```

Listing 65. C1 Implementation: GPT-5 (Score: 0.9)

```python
# DeepSeek R1: Comprehensive with proper encoding
class BioNeRF(nn.Module):
    def __init__(self, hidden_dim=256, num_freqs=10):
        self.encoded_3d_dim = 3 * (2 * num_freqs + 1)
        self.M_delta = self._build_mlp(
            self.encoded_3d_dim, hidden_dim, hidden_dim, 3)
        self.M_color = self._build_mlp(
            self.encoded_3d_dim, hidden_dim, hidden_dim, 3)
# ISSUE: Correct structure but naming inconsistency
```

Listing 66. C1 Implementation: DeepSeek R1 (Score: 0.8)

```python
# Paper2Code: Oversimplified to incorrect inputs
class BioNeRFModel(nn.Module):
    def __init__(self):
        self.mlp_initial_delta = self._build_mlp(
            self.input_dim_xyz, hidden_size, hidden_size, 3)
        self.mlp_initial_c = self._build_mlp(
            self.input_dim_dir, hidden_size, hidden_size, 3)
# ERROR: Uses different inputs for each MLP (xyz vs dir)
# Should both process positional-encoded coordinates
```

Listing 67. C1 Implementation: Paper2Code (Score: 0.3)

```python
# AutoP2C: Structure exists but fundamental error
def positional_feature_extraction(self, inputs):
    encoded_inputs = positional_encoding(inputs, 10)
    # CRITICAL ERROR: Uses raw inputs, not encoded
    h_delta = self.mlp_delta(inputs)  # Wrong!
    h_c = self.mlp_color(inputs)      # Wrong!
    return h_delta, h_c
```

Listing 68. C1 Implementation: AutoP2C (Score: 0.1)

Analysis: NERFIFY correctly implements independent parallel MLPs with exact paper specifications. GPT-5 achieves near-perfect implementation. DeepSeek R1 has correct structure but minor naming issues. Paper2Code incorrectly uses different inputs for each MLP. AutoP2C defines encoding but critically fails to use it, passing raw inputs instead.

Component C2-C6: Cognitive Filtering - All Baselines

The paper defines four cognitive filters and pre-modulation: $f_\Delta = \sigma(W_\Delta^f h_\Delta + b_\Delta^f)$, $f_c = \sigma(W_c^f h_c + b_c^f)$, $f_\psi = \sigma(W_\psi^f [h_\Delta, h_c] + b_\psi^f)$, $f_\mu = \sigma(W_\mu^f [h_\Delta, h_c] + b_\mu^f)$, and $\gamma = \tanh(W_\gamma [h_\Delta, h_c] + b_\gamma)$.

```python
# \nerfify\ : All filters properly defined in __init__
def __init__(self):
    self.W_delta_f = nn.Linear(self.hidden_dim, self.hidden_dim)
    self.W_c_f = nn.Linear(self.hidden_dim, self.hidden_dim)
    self.W_psi_f = nn.Linear(2 * self.hidden_dim, self.hidden_dim)
    self.W_mu_f = nn.Linear(2 * self.hidden_dim, self.hidden_dim)
    self.W_gamma = nn.Linear(2 * self.hidden_dim, self.hidden_dim)
def forward(self):
    f_delta = self._sigmoid(self.W_delta_f(h_delta))
    f_c = self._sigmoid(self.W_c_f(h_c))
    h_cat = torch.cat([h_delta, h_c], dim=-1)
    f_psi = self._sigmoid(self.W_psi_f(h_cat))
    f_mu = self._sigmoid(self.W_mu_f(h_cat))
    gamma = self._tanh(self.W_gamma(h_cat))
# CORRECT: All filters with proper activations
```

Listing 69. C2-C6 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Clean filter implementation
class BioNeRFField(nn.Module):
    def __init__(self):
        self.f_delta = nn.Linear(hidden, hidden)
        self.f_color = nn.Linear(hidden, hidden)
        self.f_mem = nn.Linear(hidden * 2, hidden)
        self.f_mod = nn.Linear(hidden * 2, hidden)
        self.pre_mod = nn.Linear(hidden * 2, hidden)
    def forward(self, x, d):
        f_delta = torch.sigmoid(self.f_delta(h_delta))
        f_color = torch.sigmoid(self.f_color(h_color))
        gamma = torch.tanh(self.pre_mod(h_cat))
# CORRECT: Good implementation with slight naming differences
```

Listing 70. C2-C6 Implementation: GPT-5 (Score: 0.8)

```python
# DeepSeek R1: Comprehensive filter implementation
def __init__(self):
    self.W_f_delta = nn.Linear(hidden_dim, hidden_dim)
    self.W_f_color = nn.Linear(hidden_dim, hidden_dim)
    self.W_f_psi = nn.Linear(2 * hidden_dim, hidden_dim)
    self.W_f_mu = nn.Linear(2 * hidden_dim, hidden_dim)
def forward(self):
    f_delta = torch.sigmoid(self.W_f_delta(h_delta))
    f_psi = torch.sigmoid(self.W_f_psi(h_concat))
# ISSUE: Missing W_gamma layer definition
```

Listing 71. C2-C6 Implementation: DeepSeek R1 (Score: 0.7)

```python
# Paper2Code: Partial implementation
def forward(self):
    f_delta = torch.sigmoid(self.linear_fd(h_delta))
    f_c = torch.sigmoid(self.linear_fc(h_c))
    # Concatenates but missing some filters
    h_cat = torch.cat((h_delta, h_c), dim=-1)
    gamma = torch.tanh(self.linear_gamma(h_cat))
# MISSING: f_psi and f_mu filters not properly defined
```

Listing 72. C2-C6 Implementation: Paper2Code (Score: 0.4)

```python
# AutoP2C: Fatal implementation error
def cognitive_filtering(self, features):
    h_delta, h_c = features
    # CRITICAL ERROR: Creates new layers every forward pass!
    f_delta = torch.sigmoid(
        nn.Linear(256, 256)(h_delta))  # Random weights!
    f_c = torch.sigmoid(
        nn.Linear(256, 256)(h_c))      # Random weights!
    # No gradient tracking possible
```

Listing 73. C2-C6 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY perfectly implements all cognitive filters. GPT-5 has good implementation with minor naming variations. DeepSeek R1 misses gamma layer definition. Paper2Code partially implements filters. AutoP2C has a critical flaw - creating nn.Linear layers inside forward pass results in random weights every iteration with no gradient tracking.

Component C7-C8: Memory Mechanism - All Baselines

The memory mechanism involves modulation $\mu = f_\mu \otimes \gamma$ and recursive update $\Psi = \tanh(W_\Psi(\mu + f_\psi \otimes \Psi_{prev}) + b_\Psi)$ where the memory state persists across forward passes.

```python
# \nerfify\ : Perfect stateful memory with detach
def forward(self):
    prev_mem = self._ensure_memory(N, device, dtype)
    mu = f_mu * gamma  # Element-wise modulation
    # Proper recursive update with previous memory
    Psi = self._tanh(self.W_Psi(mu + f_psi * prev_mem))
    # Detach to prevent gradient explosion
    self._memory = Psi.detach()
# CORRECT: Stateful memory with proper recursion
```

Listing 74. C7-C8 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Separate memory module approach
class MemoryModule(nn.Module):
    def forward(self, mu, fpsi):
        if self.memory is None:
            self.memory = torch.zeros_like(mu)
        updated = torch.tanh(
            self.W_psi(mu + fpsi * self.memory) + self.b_psi)
        self.memory = updated.detach()
        return updated
# CORRECT: Proper recursion, slight module complexity
```

Listing 75. C7-C8 Implementation: GPT-5 (Score: 0.8)

```python
# DeepSeek R1: Good attempt with persistence issue
def forward(self, xyz, view_dir):
```

```
    mu = f_mu * gamma  # Correct modulation
    if self.memory.shape[0] != batch_size:
        self.memory = self.memory.expand(batch_size, -1)
    memory_input = mu + (f_psi * self.memory)
    psi_new = torch.tanh(self.W_psi(memory_input))
    self.memory = psi_new
# ISSUE: Memory expansion may not persist properly
```

Listing 76. C7-C8 Implementation: DeepSeek R1 (Score: 0.6)

```
# Paper2Code: Missing recursive component
def forward(self):
    mu = f_mu * gamma
    psi_init = torch.zeros_like(mu)
    # ERROR: Always uses zeros, no memory persistence
    psi = torch.tanh(self.linear_memory_update(
        mu + (f_psi * psi_init)))  # psi_init always 0!
# MISSING: No actual memory recursion
```

Listing 77. C7-C8 Implementation: Paper2Code (Score: 0.2)

```
# AutoP2C: Multiple critical issues
def memory_updating(self, filtered_features):
    mu = f_mu * gamma
    # CRITICAL ERROR: Creates new layer here
    self.memory = torch.tanh(
        nn.Linear(256, 256)(mu + f_psi * self.memory)
    )
    # Random weights, no learning possible
```

Listing 78. C7-C8 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY implements perfect stateful memory with detach to prevent gradient issues. GPT-5 uses a correct separate module approach. DeepSeek R1 has memory persistence issues. Paper2Code fails to maintain memory state. AutoP2C creates layers dynamically, making learning impossible.

Component C9-C10: Contextual Inference and Output Heads - All Baselines

The contextual inference creates embeddings $h'_\Delta = [\Psi \otimes f_\Delta, x]$ and $h'_c = [\Psi \otimes f_c, d]$, which are processed by density head M'_Δ and color head M'_c respectively.

```
# \nerfify\ : Perfect contextual inference
def forward(self):
    ctx_delta = Psi * f_delta  # Element-wise
    ctx_c = Psi * f_c
    sigma_in = torch.cat([ctx_delta, pe], dim=-1)
    sigma_raw = self.density_head(sigma_in)
    density = trunc_exp(sigma_raw).view(*positions.shape[:-1], 1)
    rgb_in = torch.cat([ctx_c, de], dim=-1)
    rgb = self.color_head(rgb_in).view(*positions.shape[:-1], 3)
# CORRECT: Exact paper implementation
```

Listing 79. C9-C10 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Good implementation with minor differences
def forward(self):
    hpd = torch.cat([psi * f_delta, x], dim=-1)
    hpc = torch.cat([psi * f_color, d], dim=-1)
    density = self.Mp_delta(hpd)
    color = torch.sigmoid(self.Mp_color(hpc))
# CORRECT: Proper concatenation and heads
```

Listing 80. C9-C10 Implementation: GPT-5 (Score: 0.8)

```
# DeepSeek R1: Correct approach, minor issues
def forward(self):
    h_delta_prime = torch.cat([psi_new * f_delta, xyz], dim=-1)
    h_color_prime = torch.cat([psi_new * f_color, view_dir], dim=-1)
    density = self.density_head(h_delta_prime)
    color = torch.sigmoid(self.color_head(h_color_prime))
# ISSUE: Missing proper view reshaping
```

Listing 81. C9-C10 Implementation: DeepSeek R1 (Score: 0.7)

```
# Paper2Code: Simplified inference
def forward(self):
    h_delta_prime = torch.cat((psi * f_delta, 1), dim=-1)
    density = self.density_head(h_delta_prime)
    color = self.color_head(h_c_prime)
# ERROR: h_c_prime not properly defined
```

Listing 82. C9-C10 Implementation: Paper2Code (Score: 0.3)

```
# AutoP2C: Structural attempt, poor execution
def forward(self):
    density = self.heads['density'](h_delta_prime)
    color = self.heads['color'](h_c_prime)
    return density, color
# ERROR: h_delta_prime, h_c_prime undefined
# MISSING: No contextual concatenation
```

Listing 83. C9-C10 Implementation: AutoP2C (Score: 0.1)

Analysis: NERFIFY perfectly implements contextual inference with proper concatenation and reshaping. GPT-5 has good implementation with minor variations. DeepSeek R1 misses view reshaping. Paper2Code has undefined variables. AutoP2C attempts structure but lacks proper implementation.

3.1.5. Scoring Analysis

Table 28. Revised BioNeRF Implementation Scores

Method	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	Weighted Avg$_{LLM}$
NERFIFY	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.00
GPT-5	0.9	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.82
DeepSeek R1	0.8	0.7	0.7	0.7	0.7	0.7	0.7	0.6	0.7	0.7	0.70
Paper2Code	0.3	0.4	0.4	0.4	0.4	0.4	0.3	0.2	0.3	0.3	0.34
AutoP2C	0.1	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.1	0.1	0.04

3.1.6. Why Baselines Fail Despite Component Scores

GPT-5 (Score: 0.82 components, 0% trainable)

- **Strengths**: Near-perfect component implementation, clean modular code structure, proper memory handling
- **Fatal Issues**:
  - No Nerfstudio integration whatsoever
  - Missing training pipeline and data loading infrastructure
  - Standalone module without framework compatibility
- **Result**: Excellent algorithm but requires 3-5 hours of integration work to be trainable

DeepSeek R1 (Score: 0.75 components, 0% trainable)

- **Strengths**: Comprehensive implementation with all components present, proper encoding

- **Fatal Issues**:
  - Memory buffer expansion doesn't persist between batches
  - No Nerfstudio plugin structure
  - Missing critical training loop integration
- **Result**: Memory persistence issues cause degradation to vanilla NeRF behavior

Paper2Code (Score: 0.35 components, 0% trainable)

- **Strengths**: Extensive 1200+ line codebase with complete dataset loading infrastructure
- **Fatal Issues**:
  - Implements generic NeRF with BioNeRF naming conventions
  - Memory mechanism lacks recursion - always uses zeros
  - Missing true parallel pathways for M$_{d}eltaandM_{c}$
  - No Nerfstudio integration
- **Result**: Code runs but implements wrong algorithm, achieving vanilla NeRF performance

AutoP2C (Score: 0.15 components, 0% trainable)

- **Strengths**: Attempts all structural components with correct high-level organization
- **Fatal Issues**:
  - Creates nn.Linear layers in forward pass resulting in random weights every iteration
  - Positional encoding defined but not applied to inputs
  - No gradient tracking possible with dynamic layer creation
  - Fundamental PyTorch misunderstanding
- **Result**: Catastrophic implementation errors prevent any learning from occurring

3.1.7. Hyperparameter Fidelity

Table 29. Hyperparameter Implementation Accuracy

Parameter	Paper	NERFIFY	GPT-5	R1	P2C	AutoP2C
Learning rate	5e-4	✓	✓	✓	✓	×
Hidden dimensions	256	✓	✓	✓	✓	✓
Memory tensor size	8192	✓	∼	∼	×	✓
Training iterations	400k	✓	×	×	✓	×
Adam optimizer	Yes	✓	✓	✓	✓	✓
Positional frequencies	10	✓	∼	✓	✓	✓
Batch size	8192	✓	∼	×	✓	×
W Score	–	1.00	0.57	0.57	0.71	0.43

3.1.8. Conclusion

All baselines attempt to implement BioNeRF with varying levels of sophistication and understanding. GPT-5 and DeepSeek R1 achieve high component scores with nearly correct algorithmic implementations but completely lack the Nerfstudio integration necessary for training. Paper2Code produces an extensive codebase but fundamentally misunderstands the algorithm, implementing a simplified NeRF variant that misses the critical recursive memory mechanism. AutoP2C demonstrates the most severe implementation errors, creating neural network layers dynamically within the forward pass, which results in random weights every iteration and makes gradient-based learning impossible. Only NERFIFY produces immediately trainable code by combining perfect component implementation with proper framework integration as a complete Nerfstudio plugin, achieving a 100% trainable rate while all baselines remain at 0% trainable despite their varying component scores. "'

3.2. Mip-NeRF: A Multiscale Representation for Anti-Aliasing Neural Radiance Fields

3.2.1. Paper Overview

Mip-NeRF [2] addresses the critical aliasing artifacts in Neural Radiance Fields by replacing point sampling with cone tracing and standard positional encoding with integrated positional encoding (IPE). The paper's key innovation is approximating conical frustums as 3D Gaussians and computing the expected positional encoding analytically, enabling anti-aliased rendering across multiple scales with a single multiscale MLP that reduces parameters by 50% compared to NeRF.

3.2.2. Implementation Overview

Table 30. Mip-NeRF Implementation Summary Across All Baselines

Aspect	NERFIFY	GPT-5	DeepSeek R1	Paper2Code	AutoP2C
Lines of Code	563	412	287	1842	895
File Organization	Plugin	Single	Single	Multi-file	Multi-file
Conical Frustum	✓	×	✓	✓	Simplified
IPE Implementation	✓	×	✓	✓	Attempted
Single MLP	✓	✓	✓	✓	×
Trainable	✓	×	×	×	×

Note: NERFIFY always produces trainable code as a complete Nerfstudio plugin

3.2.3. Novel Components

Table 31. Novel Components in Mip-NeRF with Importance Weights

ID	Component	Weight w_i
C1	Conical Frustum Parameters (Eq. 7-8, stable formulation)	0.20
C2	Gaussian Mean Transform to World Coordinates	0.10
C3	Gaussian Covariance Transform (diagonal approximation)	0.15
C4	Integrated Positional Encoding (IPE)	0.25
C5	Single Multiscale MLP Architecture	0.15
C6	Area-Weighted Loss for Multiscale Training	0.15

3.2.4. Quantitative Metrics

3.2.5. Component-by-Component Analysis

Component C1: Conical Frustum Parameters - All Baselines

The paper derives stable reparameterized formulas (Appendix A) for computing Gaussian parameters of a conical

Table 32. Mip-NeRF Implementation Coverage Metrics

Method	C	I	M	W	$\text{Score}_{\text{LLM}}$
NERFIFY (Ours)	1.00	0.00	0.00	1.00	1.00
GPT-5	0.50	0.30	0.20	0.60	0.58
DeepSeek R1	0.67	0.17	0.16	0.67	0.58
Paper2Code	0.83	0.17	0.00	0.83	0.85
AutoP2C	0.17	0.17	0.66	0.25	0.20

frustum. Given segment endpoints t_0, t_1 and cone radius $\dot{r}$, the formulas use midpoint $t_\mu = (t_0 + t_1)/2$ and half-width $t_\delta = (t_1 - t_0)/2$ to compute mean μ_t, axial variance σ_t^2, and radial variance σ_r^2.

```
# \nerfify\ : Exact stable formulation from paper
t_mu = (t0 + t1) / 2.0
t_delta = (t1 - t0) / 2.0
mu_t = t_mu + (2.0 * t_mu * t_delta ** 2) / (3.0 *
    t_mu ** 2 + t_delta ** 2)
sigma_t_sq = (t_delta ** 2) / 3.0 - \
    (4.0 * t_delta ** 4 * (12.0 * t_mu ** 2 - t_delta
     ** 2)) / \
    (15.0 * (3.0 * t_mu ** 2 + t_delta ** 2) ** 2)
sigma_r_sq = (dot_r ** 2) * (t_mu ** 2 / 4.0 + 5.0 *
    t_delta ** 2 / 12.0 - \
    4.0 * t_delta ** 4 / (15.0 * (3.0 * t_mu ** 2 +
    t_delta ** 2)))
# CORRECT: Stable reparameterized formulas from
    Appendix A
```

Listing 84. C1 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Completely wrong formulas
def conical_frustum_to_gaussian(t0, t1, radius):
    t_mean = (3*(t1**4 - t0**4))/(4*(t1**3 - t0**3))
    t_var = (3*(t1**5 - t0**5))/(5*(t1**3 - t0**3)) -
     t_mean**2
    r_var = radius**2 * (3*(t1**5 - t0**5))/(20*(t1
    **3 - t0**3))
    return t_mean, t_var, r_var
# CRITICAL ERROR: Different derivation, numerically
    unstable
```

Listing 85. C1 Implementation: GPT-5 (Score: 0.0)

```
# DeepSeek R1: Correct stable formulation
mu_t = t_mu + (2 * t_mu * t_delta**2) / (3 * t_mu**2
    + t_delta**2)
sigma_t2 = (t_delta**2 / 3) - (4 * t_delta**4 * (12 *
     t_mu**2 - t_delta**2)) / \
    (15 * (3 * t_mu**2 + t_delta**2)**2)
sigma_r2 = r**2 * (t_mu**2 / 4 + 5 * t_delta**2 / 12
    - \
    4 * t_delta**4 / (15 * (3 * t_mu**2 + t_delta**2)
    ))
# CORRECT: Matches paper formulation
```

Listing 86. C1 Implementation: DeepSeek R1 (Score: 1.0)

```
# Paper2Code: Correct with JAX
denominator = 3 * (z_center ** 2) + z_delta ** 2 + 1e
    -8
mu_z = z_center + (2.0 * z_center * (z_delta ** 2)) /
     denominator
sigma_z_sq = (z_delta ** 2) / 3.0 - \
    (4.0 * (z_delta ** 4) * (12 * z_center ** 2 -
    z_delta ** 2)) / \
    (15.0 * (denominator ** 2) + 1e-8)
# CORRECT: Stable formulation with numerical safety
```

Listing 87. C1 Implementation: Paper2Code (Score: 1.0)

```
# AutoP2C: Missing frustum computation
# No conical frustum parameter computation found
# Uses stratified sampling without cone tracing
# MISSING: Core component entirely absent
```

Listing 88. C1 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY , DeepSeek R1, and Paper2Code correctly implement the stable reparameterized formulas. GPT-5 uses completely different formulas that are both mathematically incorrect and numerically unstable. AutoP2C entirely omits conical frustum computation.

Component C2-C3: Gaussian Mean and Covariance Transform - All Baselines

The paper transforms frustum parameters to world coordinates: $\boldsymbol{\mu} = \mathbf{o} + \mu_t \mathbf{d}$ for the mean and $\boldsymbol{\Sigma} = \sigma_t^2(\mathbf{d}\mathbf{d}^\top) + \sigma_r^2(\mathbf{I} - \mathbf{d}\mathbf{d}^\top/\|\mathbf{d}\|_2^2)$ for covariance.

```
# \nerfify\ : Efficient diagonal approximation
means = origins + mu_t[..., None] * directions  #
    Mean transform
d_outer_diag = directions ** 2
d_norm_sq = (directions ** 2).sum(dim=-1, keepdim=
    True)
covs_diag = sigma_t_sq[..., None] * d_outer_diag + \
    sigma_r_sq[..., None] * (1.0 - d_outer_diag /
    d_norm_sq)
# CORRECT: Diagonal covariance for efficiency
```

Listing 89. C2-C3 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Full covariance matrix (inefficient)
mu = rays_o[..., None, :] + mu_z[..., None] * rays_d
    [..., None, :]
covs = (rays_d[..., None] * rays_d[..., None, :] *
    t_var[..., None, None]
    + jnp.eye(3) * r_var[..., None, None])
# ISSUE: Computes full 3x3 matrix, inefficient but
    mathematically correct
```

Listing 90. C2-C3 Implementation: GPT-5 (Score: 0.5)

```
# Paper2Code: Correct diagonal implementation
mu = rays_o_exp + jnp.expand_dims(mu_z, axis=-1) *
    rays_d_exp
r_squared = rays_d ** 2
cov_diag = jnp.expand_dims(sigma_z_sq, axis=-1) *
    r_squared_exp + \
    jnp.expand_dims(sigma_r_sq, axis=-1) * (1.0 -
    r_squared_exp)
# CORRECT: Efficient diagonal form
```

Listing 91. C2-C3 Implementation: Paper2Code (Score: 1.0)

Analysis: Most baselines correctly implement the transforms. GPT-5 computes the full covariance matrix instead of just the diagonal, which is mathematically correct but computationally inefficient.

Component C4: Integrated Positional Encoding (IPE) - All Baselines

The core innovation computes expected positional encoding over a Gaussian: $\mathbb{E}[\sin(x)] = \sin(\mu)\exp(-\sigma^2/2)$ and $\mathbb{E}[\cos(x)] = \cos(\mu)\exp(-\sigma^2/2)$ for frequency bands $2^0, 2^1, ..., 2^{L-1}$.

```
# \nerfify\ : Perfect IPE implementation
scales = 2.0 ** torch.arange(self.
    position_encoding_max_degree)
means_scaled = means[..., :, None] * scales
variance_scales = (scales ** 2)
covs_scaled = covs_diag[..., :, None] *
    variance_scales
damping = torch.exp(-0.5 * covs_scaled)
sin_features = torch.sin(means_scaled) * damping
cos_features = torch.cos(means_scaled) * damping
features = torch.cat([sin_features, cos_features],
    dim=-1)
# CORRECT: Exact formulation from paper
```

Listing 92. C4 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Multiple critical errors
freqs = 2.0 ** jnp.arange(max_freq_log2)
scales = jnp.pi * freqs[None, None, :]  # ERROR:
    Extra pi!
diag = jnp.stack([cov[..., i, i] for i in range(3)],
    axis=-1)
var = diag[..., None, :] * scales[..., None] ** 2
# CRITICAL ERROR: Wrong frequency scaling, dimension
    mismatches
```

Listing 93. C4 Implementation: GPT-5 (Score: 0.2)

```
# Paper2Code: Correct JAX implementation
freq_bands = 2.0 ** jnp.arange(L, dtype=jnp.float32)
args = mu_exp * freq_bands
damping = jnp.exp(-0.5 * (cov_exp * (freq_bands ** 2)
    ))
sin_features = jnp.sin(args) * damping
cos_features = jnp.cos(args) * damping
# CORRECT: Proper IPE computation
```

Listing 94. C4 Implementation: Paper2Code (Score: 1.0)

Analysis: NERFIFY , DeepSeek R1, and Paper2Code correctly implement IPE. GPT-5 has multiple errors including extra π multiplication and dimension mismatches. AutoP2C attempts IPE but with incorrect structure.

Component C5: Single MLP Architecture - All Baselines

Mip-NeRF uses a single multiscale MLP instead of separate coarse/fine networks, reducing parameters by 50

```
# \nerfify\ : Single MLP with hierarchical sampling
    in Nerfstudio
class MipNeRFModel(Model):
    def populate_modules(self):
        self.field = MipNeRFField()  # Single field
        self.sampler_uniform = UniformSampler(
    num_samples=128)
        self.sampler_pdf = PDFSampler(num_samples
    =128, include_original=False)
# CORRECT: Single MLP queried hierarchically
```

Listing 95. C5 Implementation: NERFIFY (Score: 1.0)

```
# AutoP2C: Uses two separate MLPs
coarse_out = self.model.apply({"params": self.state.
    params}, ray_batch_coarse)
# ... hierarchical sampling ...
fine_out = state.apply_fn({"params": params},
    ray_batch_fine)
# ERROR: Separate coarse/fine networks instead of
    single MLP
```

Listing 96. C5 Implementation: AutoP2C (Score: 0.0)

Analysis: Most baselines correctly implement a single MLP. AutoP2C incorrectly uses hierarchical two-network architecture from original NeRF.

Component C6: Area-Weighted Loss - All Baselines

For multiscale training, pixel losses are weighted by their footprint area.

```
# \nerfify\ : Full area-weighted loss support
if self.config.area_loss_mult > 0:
    pixel_area = ray_bundle.pixel_area
    loss = loss + pixel_area.squeeze() * self.config.
    area_loss_mult
# CORRECT: Proper multiscale weighting
```

Listing 97. C6 Implementation: NERFIFY (Score: 1.0)

```
# Paper2Code: Partial implementation
def area_weighted_loss(pred_rgb, target_rgb,
    pixel_area, lambda=0.1):
    diff = (pred_rgb - target_rgb)**2
    return lambda * jnp.mean(diff * pixel_area[...,
    None])
# ISSUE: Defined but not integrated into training
    loop
```

Listing 98. C6 Implementation: Paper2Code (Score: 0.5)

Analysis: Only NERFIFY fully implements area-weighted loss. Paper2Code defines it but doesn't integrate it. Others omit it entirely.

3.2.6. Scoring Analysis

Table 33. Mip-NeRF Implementation Scores

Method	C1	C2	C3	C4	C5	C6	Weighted Avg$_{\text{LLM}}$
NERFIFY	1.0	1.0	1.0	1.0	1.0	1.0	1.00
GPT-5	0.0	1.0	0.5	0.2	1.0	0.0	0.58
DeepSeek R1	1.0	1.0	1.0	1.0	0.5	0.0	0.58
Paper2Code	1.0	1.0	1.0	1.0	1.0	0.5	0.85
AutoP2C	0.0	0.5	0.5	0.3	0.0	0.0	0.20

3.2.7. Why Baselines Fail Despite Component Scores

GPT-5 (Score: 0.58 components, 0% trainable)

- **Strengths**: Implements single MLP architecture and basic transforms
- **Fatal Issues**:
  - Wrong conical frustum formulas cause NaN gradients
  - IPE implementation has critical errors (extra π, dimension mismatches)
  - JAX code incompatible with Nerfstudio benchmark
- **Result**: Non-trainable due to fundamental mathematical errors

DeepSeek R1 (Score: 0.58 components, 0% trainable)

- **Strengths**: Correct mathematical formulations for frustum and IPE
- **Fatal Issues**:
  - No Nerfstudio integration (standalone PyTorch)
  - Missing critical training infrastructure

  - Simplified hierarchical sampling without proposal network
- **Result**: Cannot be executed in benchmark environment

Paper2Code (Score: 0.85 components, 0% trainable)

- **Strengths**: Most accurate mathematical implementation in JAX/Flax
- **Fatal Issues**:
  - Wrong framework (JAX instead of PyTorch/Nerfstudio)
  - Incompatible API prevents benchmark evaluation
  - Requires complete infrastructure rewrite
- **Result**: Excellent standalone code but non-trainable in benchmark

AutoP2C (Score: 0.20 components, 0% trainable)

- **Strengths**: Attempts basic NeRF structure
- **Fatal Issues**:
  - Missing conical frustum computation entirely
  - Uses two MLPs instead of single architecture
  - Incomplete Nerfstudio integration
  - Many placeholder functions
- **Result**: Fundamentally incomplete implementation

3.2.8. Hyperparameter Fidelity

Table 34. Hyperparameter Implementation Accuracy

Parameter	Paper	NERFIFY	GPT-5	R1	P2C	AutoP2C
PE frequencies (L)	16	✓	✓	✓	✓	×
Direction encoding	4	✓	×	✓	×	×
Hidden dimension	256	✓	✓	✓	✓	✓
Learning rate	$5e^{-4}$	✓	✓	✓	✓	×
Batch size	4096	✓	×	×	✓	✓
W Score	–	1.00	0.60	0.67	0.83	0.25

3.2.9. Conclusion

All baselines attempt to implement Mip-NeRF with varying sophistication. Paper2Code achieves the highest component accuracy (0.85) with correct mathematical implementations but uses JAX/Flax incompatible with Nerfstudio. GPT-5 and DeepSeek R1 both score 0.58 but fail differently - GPT-5 has fundamental mathematical errors while R1 lacks integration. AutoP2C scores lowest (0.20) missing core components entirely. Only NERFIFY produces immediately trainable code as a complete Nerfstudio plugin, achieving 100% trainable rate while all baselines achieve 0%.

3.3. PyNeRF: Pyramidal Neural Radiance Fields

3.3.1. Paper Overview

PyNeRF introduces a pyramidal multiscale architecture for neural radiance fields that addresses aliasing artifacts through dynamic resolution selection. The method maintains $L = 8$ pyramid levels with geometrically increasing resolutions $N_l = N_0 \times s^l$ (where $N_0 = 16$ and $s = 2$), selecting the appropriate level for each sample based on its projected pixel area $P(x)$ using the mapping function $M(P) = \log_s(P/N_0)$. The key innovation lies in per-sample level selection with linear interpolation between adjacent pyramid levels to achieve smooth transitions.

3.3.2. Implementation Overview

Table 35. PyNeRF Implementation Summary Across All Baselines

Aspect	NERFIFY	GPT-5	DeepSeek R1	Paper2Code	AutoP2C
Lines of Code	420	185	380	1250	450
File Organization	Plugin	Modular	Single	Multi-file	Multi-file
Pyramid Architecture	✓	✓	✓	✓	×
Level Selection	✓	✓	Partial	✓	×
Interpolation	✓	✓	Partial	✓	×
Projected Area	✓	∼	×	×	×
Nerfstudio Integration	✓	×	×	×	×
Trainable	✓	×	×	×	×

Note: NERFIFY always produces trainable code as a complete Nerfstudio plugin

3.3.3. Novel Components

Table 36. Novel Components in PyNeRF with Importance Weights

ID	Component	Weight w_i
C1	Pyramidal architecture with L levels	0.15
C2	Per-sample level selection based on projected area	0.20
C3	Linear interpolation between levels	0.15
C4	Shared multiresolution features	0.10
C5	Per-level tiny MLPs (64/128 hidden)	0.10
C6	Projected area computation $P(x) = pixel\_area \times t^2$	0.10
C7	Adaptive level supervision	0.05
C8	Modern backbone integration (iNGP/TensoRF)	0.05
C9	Multiple interpolation modes (linear/Laplacian)	0.05
C10	Specific hyperparameters (8 levels, s=2, N0=16)	0.05

3.3.4. Quantitative Metrics

Table 37. PyNeRF Implementation Coverage Metrics

Method	C	I	M	W	$\text{Score}_{\text{LLM}}$
NERFIFY (Ours)	0.95	0.05	0.00	1.00	0.97
GPT-5	0.40	0.30	0.30	0.80	0.52
DeepSeek R1	0.30	0.60	0.10	0.70	0.68
Paper2Code	0.50	0.30	0.20	0.60	0.58
AutoP2C	0.00	0.10	0.90	0.00	0.03

3.3.5. Component-by-Component Analysis

Component C1: Pyramidal Architecture with L Levels - All Baselines The paper specifies creating $L = 8$ pyramid levels where each level l has resolution $N_l = N_0 \times s^l$ with separate processing heads for each level.

```
# \nerfify\ : ModuleList of per-level heads
self.density_heads = nn.ModuleList([
    MLP(in_dim=self.feature_dim, num_layers=1,
        layer_width=density_hidden_dim, out_dim=1)
    for _ in range(self.num_pyramid_levels)
])
```

```
self.color_heads = nn.ModuleList([
    MLP(in_dim=color_in_dim, num_layers=2,
        layer_width=color_hidden_dim, out_dim=3)
    for _ in range(self.num_pyramid_levels)
])
# CORRECT: Exactly L separate heads as specified
```

Listing 99. C1 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: List comprehension for levels
models = [PyNeRFLevel(enc, cfg) for _ in range(cfg.
    n_levels)]
pipeline = PyNeRFPipeline(models, cfg).cuda()
# CORRECT: Creates pyramid structure but lacks
    integration
```

Listing 100. C1 Implementation: GPT-5 (Score: 1.0)

```
# DeepSeek R1: ModuleList with incomplete
    implementation
self.levels = nn.ModuleList()
for i in range(num_levels):
    level = PyNeRFLevel(base_nerf_config, level_idx=i
    ,
                        use_shared_features=
    use_shared_features)
    self.levels.append(level)
# ISSUE: Structure correct but PyNeRFLevel incomplete
```

Listing 101. C1 Implementation: DeepSeek R1 (Score: 1.0)

```
# Paper2Code: Detailed pyramid construction
self.heads = nn.ModuleList()
for level in range(self.pyramid_levels):
    effective_resolution = self.base_resolution * (
    self.scale_factor ** level)
    head = PyNeRFHead(input_dim=num_features,
                      effective_resolution=
    effective_resolution)
    self.heads.append(head)
# CORRECT: Proper pyramid with effective resolutions
```

Listing 102. C1 Implementation: Paper2Code (Score: 1.0)

```
# AutoP2C: Single MLP for all samples
self.density_mlp = nn.Sequential(
    nn.Linear(2 * 10 * 3, 64), nn.ReLU(), nn.Linear
    (64, 1))
self.color_mlp = nn.Sequential(
    nn.Linear(2 * 10 * 6 + 64, 128), nn.ReLU(),
    nn.Linear(128, 128), nn.ReLU(), nn.Linear(128, 3)
    )
# CRITICAL ERROR: No pyramid structure at all
```

Listing 103. C1 Implementation: AutoP2C (Score: 0.0)

Analysis: All methods except AutoP2C correctly implement the pyramid architecture. AutoP2C completely misses this core concept.

Component C2: Per-Sample Level Selection - All Baselines The paper requires computing $M(P) = \log_s(P/N_0)$ and selecting level $l = \lceil M(P) \rceil$ with interpolation weight $w = l - M(P)$.

```
# \nerfify\ : Exact paper formulation
t_mid = (ray_samples.frustums.starts + ray_samples.
    frustums.ends) * 0.5
pixel_area = ray_samples.frustums.pixel_area
proj_area = pixel_area * torch.clamp(t_mid, min=0.0)
    ** 2 + 1e-12
s = torch.tensor(self.pyramid_scale_factor, device=
    proj_area.device)
M = torch.log(proj_area / max(self.base_proj_area, 1e
    -12)) / torch.log(s)
l = torch.clamp(torch.ceil(M), min=0.0, max=float(
    self.num_pyramid_levels - 1))
w = l - M  # Interpolation weight
# CORRECT: Matches paper equations exactly
```

Listing 104. C2 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Clean implementation
def compute_level(Px, N0, s):
    M = torch.log2(Px / N0) / torch.log2(torch.tensor
    (s))
    l = torch.clamp(torch.ceil(M), 0, None)
    w = l - M
    return l.long(), w
# CORRECT: Proper level selection logic
```

Listing 105. C2 Implementation: GPT-5 (Score: 1.0)

```
# DeepSeek R1: Incomplete with heuristics
def compute_sample_scale(self, rays, sample_positions
    ):
    sample_distances = torch.norm(sample_positions -
    rays['origins'].unsqueeze(1), dim=-1)
    pixel_areas = sample_distances ** 2 * 0.01  #
    heuristic scaling
    return pixel_areas.unsqueeze(-1).expand_as(
    sample_distances)
M = torch.log(scales / self.base_resolution) / np.log
    (self.scale_factor)
levels = torch.clamp(torch.ceil(M), 0, self.
    num_levels - 1).long()
# ERROR: Heuristic area computation, not paper
    formula
```

Listing 106. C2 Implementation: DeepSeek R1 (Score: 0.6)

```
# Paper2Code: Correct logic, wrong area formula
def compute_integration_area(self, depths):
    integration_area = (depths / self.focal) ** 2
    return integration_area
M_value = torch.log(integration_area / (base_res +
    eps)) / math.log(scale + eps)
l_target = torch.clamp(torch.ceil(M_value), min=0,
    max=self.pyramid_levels - 1)
w = l_target - M_value
# CRITICAL ERROR: Wrong formula - should be
    pixel_area * t^2
```

Listing 107. C2 Implementation: Paper2Code (Score: 0.8)

```
# AutoP2C: Placeholder that doesn't work
def compute_projected_area(self, x):
    # Placeholder for actual projected area
    computation
    return torch.ones_like(x[..., 0])
# CRITICAL ERROR: Always returns 1.0, no level
    selection
```

Listing 108. C2 Implementation: AutoP2C (Score: 0.0)

Analysis: Only NERFIFY and GPT-5 implement correct level selection. Paper2Code's critical error in area computation undermines the entire method.

Component C3: Linear Interpolation Between Levels - All Baselines The paper specifies linear interpolation: $\sigma, \mathbf{c} = wf_l(\mathbf{x}, \mathbf{d}) + (1 - w)f_{l-1}(\mathbf{x}, \mathbf{d})$.

```python
# \nerfify\ : Perfect interpolation implementation
rgb_main = torch.gather(rgb_all, 0, idx_main).squeeze(0)
rgb_prev = torch.gather(rgb_all, 0, idx_prev).squeeze(0)
if self.interp_mode == "interpolate":
    rgb = w * rgb_main + (1.0 - w) * rgb_prev
else:  # Laplacian mode
    rgb_cum = torch.cumsum(rgb_all, dim=0)
    rgb = torch.gather(rgb_cum, 0, idx_main).squeeze(0)
# CORRECT: Exact Equation 5 with multiple modes
```

Listing 109. C3 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Clean interpolation
def forward(self, x, d, deltas, Px):
    l, w = compute_level(Px, self.cfg.base_resolution, self.cfg.scale_factor)
    c1, s1 = self.models[l-1](x, d)
    c2, s2 = self.models[l](x, d)
    c = w.unsqueeze(-1) * c2 + (1 - w).unsqueeze(-1) * c1
    sigma = w * s2 + (1 - w) * s1
    return volume_render(c, sigma, deltas)
# CORRECT: Proper linear interpolation
```

Listing 110. C3 Implementation: GPT-5 (Score: 1.0)

```python
# DeepSeek R1: Partial implementation
if self.interpolation_method == "linear" and level_idx > 0:
    prev_output = self.levels[level_idx - 1](level_samples, level_dirs)
    level_weights = weights[mask].unsqueeze(-1)
    level_output['density'] = (level_weights * level_output['density'] +
                               (1 - level_weights) * prev_output['density'])
    # Note: Color interpolation similar but omitted in provided code
# ISSUE: Incomplete - missing color interpolation details
```

Listing 111. C3 Implementation: DeepSeek R1 (Score: 0.8)

```python
# Paper2Code: Correct interpolation logic
if level_int == 0:
    density, color = self.heads[0](x_level, d_level)
else:
    density_upper, color_upper = self.heads[level_int](x_level, d_level)
    density_lower, color_lower = self.heads[level_int - 1](x_level, d_level)
    density = w_level * density_upper + (1 - w_level) * density_lower
    color = w_level * color_upper + (1 - w_level) * color_lower
# CORRECT: Proper interpolation between adjacent levels
```

Listing 112. C3 Implementation: Paper2Code (Score: 1.0)

```python
# AutoP2C: Nonsensical interpolation
if self.multiscale_sampling:
    # Linearly interpolate between the outputs of two closest hierarchy levels
    if self.multiscale_sampling:
        density = w * density + (1 - w) * density
        color = w * color + (1 - w) * color
# CRITICAL ERROR: Interpolates with itself - no effect
```

Listing 113. C3 Implementation: AutoP2C (Score: 0.1)

Analysis: NERFIFY , GPT-5, and Paper2Code correctly implement interpolation. AutoP2C's self-interpolation is a fundamental misunderstanding.

Component C4: Shared Multiresolution Features - All Baselines The paper suggests reusing multi-resolution features across levels when the backbone already has a scale hierarchy.

```python
# \nerfify\ : Shared hash encoding across all levels
self.hash_map = HashEncoding(
    num_levels=num_hash_levels, min_res=min_res, max_res=max_res,
    log2_hashmap_size=log2_hashmap_size, features_per_level=features_per_level)
# All pyramid levels query the same shared encoding:
feats = self.hash_map(pos_flat)  # Used by all density/color heads
# CORRECT: Single shared encoding as specified
```

Listing 114. C4 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Same encoder reference for all levels
enc = HashEncoder(cfg.n_levels, cfg.features_per_level,
                  cfg.base_resolution, cfg.scale_factor, cfg.hash_size)
models = [PyNeRFLevel(enc, cfg) for _ in range(cfg.n_levels)]
# ERROR: All levels share same encoder instance - problematic
```

Listing 115. C4 Implementation: GPT-5 (Score: 0.2)

```python
# DeepSeek R1: Configurable shared features
def __init__(self, use_shared_features: bool = True):
    if not use_shared_features:
        self.feature_grid = self._create_feature_grid(base_config, level_idx)
    else:
        self.feature_grid = None  # Will use shared features from base NeRF
# ISSUE: Structure present but implementation incomplete
```

Listing 116. C4 Implementation: DeepSeek R1 (Score: 0.6)

```python
# Paper2Code: Single shared grid
self.shared_grid = True
self.feature_grid = FeatureGrid(resolution=self.base_resolution,
                                num_features=num_features,
                                hash_table_size=hash_table_size)
# Each head references the same grid
head = PyNeRFHead(feature_grid=self.feature_grid)
# ISSUE: Simple hash table, not true multi-resolution
```

Listing 117. C4 Implementation: Paper2Code (Score: 0.6)

```python
# AutoP2C: No feature sharing concept
self.positional_encoding = positional_encoding
x_encoded = self.apply_positional_encoding(x)
# MISSING: No shared features, uses positional encoding only
```

Listing 118. C4 Implementation: AutoP2C (Score: 0.0)

Analysis: Only NERFIFY implements proper shared multi-resolution features. Others either misunderstand or omit this optimization.

Component C5: Per-Level Tiny MLPs - All Baselines The paper specifies tiny MLPs per level: 64-channel density MLP (1 hidden layer) and 128-channel color MLP (2 hidden layers).

```
# \nerfify\ : Exact specifications
self.density_heads = nn.ModuleList([
    MLP(in_dim=self.feature_dim, num_layers=1,
        layer_width=density_hidden_dim,  # 64
        out_dim=1, activation=nn.ReLU())
    for _ in range(self.num_pyramid_levels)])
self.color_heads = nn.ModuleList([
    MLP(in_dim=color_in_dim, num_layers=2,
        layer_width=color_hidden_dim,  # 128
        out_dim=3, activation=nn.ReLU(),
     out_activation=nn.Sigmoid())
    for _ in range(self.num_pyramid_levels)])
# CORRECT: Matches paper architecture exactly
```

Listing 119. C5 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Simplified MLPs
class DensityMLP(nn.Module):
    def __init__(self, in_dim, hidden_dim):
        self.fc = nn.Sequential(nn.Linear(in_dim,
     hidden_dim),
                                nn.ReLU(), nn.Linear(
     hidden_dim, 1))
class ColorMLP(nn.Module):
    def __init__(self, in_dim, hidden_dim):
        self.fc = nn.Sequential(nn.Linear(in_dim + 3,
      hidden_dim),
                                nn.ReLU(), nn.Linear(
     hidden_dim, 3))
# ISSUE: Missing second hidden layer for color
```

Listing 120. C5 Implementation: GPT-5 (Score: 0.6)

```
# DeepSeek R1: TinyCUDANN networks
self.density_mlp = tcnn.Network(
    n_input_dims=self._get_input_dims(config),
     n_output_dims=1,
    network_config={"otype": "FullyFusedMLP", "
     activation": "ReLU",
                    "n_neurons": 64, "n_hidden_layers"
     : 1})
self.color_mlp = tcnn.Network(
    n_input_dims=self._get_input_dims(config) + 3,
     n_output_dims=3,
    network_config={"otype": "FullyFusedMLP", "
     n_neurons": 128,
                    "n_hidden_layers": 2})
# ISSUE: Using external library, not pure PyTorch
```

Listing 121. C5 Implementation: DeepSeek R1 (Score: 0.6)

```
# Paper2Code: Correct MLP specifications
density_layers = []
for hidden_dim in density_hidden_dims:  # [64]
    density_layers.append(nn.Linear(current_dim,
     hidden_dim))
    density_layers.append(nn.ReLU())
self.density_mlp = nn.Sequential(*density_layers)
# Color MLP with 2 hidden layers [128, 128]
for hidden_dim in color_hidden_dims:
    color_layers.append(nn.Linear(current_dim,
     hidden_dim))
    color_layers.append(nn.ReLU())
# CORRECT: Proper layer counts and dimensions
```

Listing 122. C5 Implementation: Paper2Code (Score: 1.0)

```
# AutoP2C: Single MLPs for everything
self.density_mlp = nn.Sequential(
    nn.Linear(2 * 10 * 3, 64), nn.ReLU(), nn.Linear
     (64, 1))
# MISSING: No per-level MLPs, just one shared
```

Listing 123. C5 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY and Paper2Code match specifications exactly. GPT-5 and DeepSeek R1 have the right dimensions but implementation issues.

Component C6: Projected Area Computation - All Baselines The paper specifies computing $P(x) = pixel\_area \times t^2$ where t is the sample distance along the ray.

```
# \nerfify\ : Exact paper formula
t_mid = (ray_samples.frustums.starts + ray_samples.
     frustums.ends) * 0.5
pixel_area = ray_samples.frustums.pixel_area
if pixel_area.shape != t_mid.shape:
    pixel_area = pixel_area.expand_as(t_mid)
proj_area = pixel_area * torch.clamp(t_mid, min=0.0)
     ** 2 + 1e-12
# CORRECT: P(x) = pixel_area * t^2 as specified
```

Listing 124. C6 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Simplified approach
# In dummy loader:
torch.ones(1_024, 1).cuda()*0.5  # projected area
# Missing actual computation, uses placeholder
# ISSUE: No real projected area calculation
```

Listing 125. C6 Implementation: GPT-5 (Score: 0.6)

```
# DeepSeek R1: Heuristic approximation
sample_distances = torch.norm(sample_positions - rays
     ['origins'].unsqueeze(1), dim=-1)
# Approximate pixel area as distance^2 * solid_angle
pixel_areas = sample_distances ** 2 * 0.01  #
     heuristic scaling
# ERROR: Heuristic formula, not paper specification
```

Listing 126. C6 Implementation: DeepSeek R1 (Score: 0.6)

```
# Paper2Code: WRONG FORMULA
def compute_integration_area(self, depths):
    depths = depths.to(torch.float32)
    integration_area = (depths / self.focal) ** 2
    return integration_area
# CRITICAL ERROR: Should be pixel_area * t^2, not (t/
     f)^2
```

Listing 127. C6 Implementation: Paper2Code (Score: 0.2)

```
# AutoP2C: No implementation
def compute_projected_area(self, x):
    # Placeholder for actual projected area
     computation
    return torch.ones_like(x[..., 0])
# CRITICAL ERROR: Always returns 1.0
```

Listing 128. C6 Implementation: AutoP2C (Score: 0.0)

Analysis: Only NERFIFY implements the correct formula. Paper2Code's error here is particularly damaging as it affects all level selections.

Component C7: Adaptive Level Supervision - All Baselines The paper mentions maintaining an auxiliary structure to track supervised levels during training.

```
# \nerfify\ : Occupancy grid from parent class
# Inherits from NerfactoModel which includes:
# self.occupancy_grid (from parent)
# self.proposal_networks (for sampling)
# Partial implementation through inheritance
# ISSUE: Not explicitly tracking supervised levels
```

Listing 129. C7 Implementation: NERFIFY (Score: 0.6)

```
# GPT-5: No adaptive supervision
# No occupancy grid or level tracking
# MISSING: Feature not implemented
```

Listing 130. C7 Implementation: GPT-5 (Score: 0.0)

```
# DeepSeek R1: Structure present
self.occupancy_grid = None
self.supervised_levels = None
# For tracking supervised levels during training
# ISSUE: Declared but never used
```

Listing 131. C7 Implementation: DeepSeek R1 (Score: 0.4)

```
# Paper2Code: No adaptive supervision
# MISSING: No occupancy grid or adaptive training
```

Listing 132. C7 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: No supervision concepts
# MISSING: Feature not implemented
```

Listing 133. C7 Implementation: AutoP2C (Score: 0.0)

Analysis: None of the baselines fully implement adaptive supervision. NERFIFY partially inherits it from Nerfacto.

Component C8: Modern Backbone Integration - All Baselines The paper mentions integration with modern grid-based NeRFs like iNGP, K-Planes, TensoRF.

```
# \nerfify\ : Full Nerfstudio integration
from nerfstudio.models.nerfacto import NerfactoModel,
      NerfactoModelConfig
class PynerfModel(NerfactoModel):
    """Nerfacto-based model with PyNeRF multiscale
     field."""
    def populate_modules(self):
        super().populate_modules()
        # Reuse scene contraction from Nerfacto
        spatial_distortion = getattr(self.field, "
     spatial_distortion", None)
# CORRECT: Complete framework integration
```

Listing 134. C8 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: No framework integration
# Standalone implementation
# MISSING: Cannot leverage existing infrastructure
```

Listing 135. C8 Implementation: GPT-5 (Score: 0.0)

```
# DeepSeek R1: Multiple backbone options
def _create_base_nerf(self, config):
    nerf_type = config.get('type', 'ingp')
    if nerf_type == 'ingp':
        return InstantNGPBackbone(config)
    elif nerf_type == 'tensorf':
        return TensoRFBackbone(config)
    elif nerf_type == 'kplanes':
        return KPlanesBackbone(config)
# ISSUE: Stubs only, not implemented
```

Listing 136. C8 Implementation: DeepSeek R1 (Score: 0.8)

```
# Paper2Code: No backbone integration
# Custom standalone implementation
# MISSING: No framework compatibility
```

Listing 137. C8 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: No backbone concepts
# MISSING: Feature not implemented
```

Listing 138. C8 Implementation: AutoP2C (Score: 0.0)

Analysis: Only NERFIFY achieves real framework integration. DeepSeek R1 has the structure but lacks implementation.

Component C9: Multiple Interpolation Modes - All Baselines The paper describes both linear interpolation (Equation 5) and Laplacian pyramid summation (Equation 4).

```
# \nerfify\ : Both interpolation modes
if self.interp_mode == "sum":
    # Laplacian pyramid style (Equation 4)
    sigma_cum = torch.cumsum(sigma_all, dim=0)
    sigma = torch.gather(sigma_cum, 0, idx_main).
     squeeze(0)
else:
    # Linear interpolation (Equation 5)
    sigma = w * sigma_main + (1.0 - w) * sigma_prev
# CORRECT: Both modes implemented
```

Listing 139. C9 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Only linear interpolation
c = w.unsqueeze(-1) * c2 + (1 - w).unsqueeze(-1) * c1
# MISSING: No Laplacian mode
```

Listing 140. C9 Implementation: GPT-5 (Score: 0.0)

```
# DeepSeek R1: Mode selection present
def __init__(self, interpolation_method: str = "
     linear"):
    self.interpolation_method = interpolation_method
# ISSUE: Only linear mode implemented
```

Listing 141. C9 Implementation: DeepSeek R1 (Score: 0.4)

```
# Paper2Code: No mode selection
# Only implements linear interpolation
# MISSING: No Laplacian option
```

Listing 142. C9 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: No interpolation modes
# MISSING: Feature not implemented
```

Listing 143. C9 Implementation: AutoP2C (Score: 0.0)

Analysis: Only NERFIFY implements both interpolation modes as described in the paper.

Component C10: Specific Hyperparameters - All Baselines The paper specifies: 8 pyramid levels, scale factor 2, base resolution 16, 4 features per level, hash table size 2^{20}.

```python
# \nerfify\ : All hyperparameters correct
num_pyramid_levels: int = 8
pyramid_scale_factor: float = 2.0
features_per_level: int = 4
log2_hashmap_size: int = 20
density_hidden_dim: int = 64
color_hidden_dim: int = 128
max_num_iterations=20000
train_num_rays_per_batch=8192
# CORRECT: Matches paper exactly
```

Listing 144. C10 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Configuration matches
@dataclass
class PyNeRFConfig:
    n_levels: int = 8
    base_resolution: int = 16
    scale_factor: float = 2.0
    features_per_level: int = 4
    hash_size: int = 2**20
    mlp_hidden_dim: int = 64
    color_hidden_dim: int = 128
# CORRECT: All parameters specified
```

Listing 145. C10 Implementation: GPT-5 (Score: 1.0)

```python
# DeepSeek R1: Complete configuration
'pyramid': {'num_levels': 8, 'scale_factor': 2.0,
            'base_resolution': 16, '
     use_shared_features': True},
'base_nerf': {'log2_hash_size': 19, '
     features_per_level': 4}
'training': {'batch_size': 8192}
# CORRECT: Parameters match paper
```

Listing 146. C10 Implementation: DeepSeek R1 (Score: 1.0)

```python
# Paper2Code: Most parameters correct
self.pyramid_levels: int = int(model_config.get("
     pyramid_levels", 8))
self.base_resolution: int = int(model_config.get("
     base_resolution", 256))  # Wrong!
self.scale_factor: int = int(model_config.get("
     scale_factor", 2))
# ERROR: Base resolution should be 16, not 256
```

Listing 147. C10 Implementation: Paper2Code (Score: 0.6)

```python
# AutoP2C: Wrong parameters
hierarchy_levels=8  # Only this is correct
base_resolution=1  # Wrong! Should be 16
scaling_factor=2
# CRITICAL ERROR: Multiple wrong values
```

Listing 148. C10 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY , GPT-5, and DeepSeek R1 correctly specify all hyperparameters. Paper2Code and AutoP2C have critical errors in base resolution.

Table 38. PyNeRF Implementation Scores

Method	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	Weighted Avg$_{LLM}$
NERFIFY	1.0	1.0	1.0	1.0	1.0	1.0	0.6	1.0	1.0	1.0	0.97
GPT-5	1.0	1.0	1.0	0.2	0.6	0.6	0.0	0.0	0.0	1.0	0.52
DeepSeek R1	1.0	0.6	0.8	0.6	0.6	0.6	0.4	0.8	0.4	1.0	0.68
Paper2Code	1.0	0.8	1.0	0.6	1.0	0.2	0.0	0.0	0.0	0.6	0.58
AutoP2C	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.03

3.3.6. Scoring Analysis

3.3.7. Why Baselines Fail Despite Component Scores

GPT-5 (Score: 0.52 components, 0% trainable)

- **Strengths**: Clean modular implementation with correct pyramid structure and interpolation logic
- **Fatal Issues**:
  - No Nerfstudio integration - standalone code cannot leverage existing infrastructure
  - Missing training loop and data loading pipeline
  - Dummy data loader prevents actual training
- **Result**: Code compiles but cannot train on real data

DeepSeek R1 (Score: 0.68 components, 0% trainable)

- **Strengths**: Comprehensive structure with multi-agent architecture consideration
- **Fatal Issues**:
  - Many methods left unimplemented with pass statements
  - Heuristic projected area computation (sample_distances**2 * 0.01)
  - Incomplete PyNeRFLevel forward pass
- **Result**: Structural skeleton without functional implementation

Paper2Code (Score: 0.58 components, 0% trainable)

- **Strengths**: Most complete baseline with proper pyramid architecture and detailed implementation
- **Fatal Issues**:
  - Critical error in projected area formula: uses (depth/focal)$^2 instead of pixel\_area * t^2$
  - No framework integration - standalone implementation
  - Missing adaptive supervision and occupancy grid
- **Result**: Despite 1250 lines of code, mathematical errors prevent convergence

AutoP2C (Score: 0.03 components, 0% trainable)

- **Strengths**: Attempts to create training infrastructure
- **Fatal Issues**:
  - Complete architectural misunderstanding - no pyramid structure
  - Placeholder functions that always return constants
  - Import errors for non-existent modules (colmap Python module)
  - Self-interpolation bug (interpolates values with themselves)
- **Result**: Complete failure to understand the paper's core concepts

Table 39. Hyperparameter Implementation Accuracy

Parameter	Paper	NERFIFY	GPT-5	R1	P2C	AutoP2C
Pyramid Levels (L)	8	✓	✓	✓	✓	×
Scale Factor (s)	2	✓	✓	✓	✓	×
Base Resolution	16	✓	✓	✓	∼	×
Features per Level	4	✓	✓	∼	✓	×
Hash Table Size	2^{20}	✓	✓	×	✓	×
Density MLP Hidden	64	✓	✓	✓	✓	✓
Color MLP Hidden	128	✓	✓	✓	✓	✓
Training Iterations	20,000	✓	✓	×	✓	✓
Batch Size	8,192	✓	∼	✓	✓	✓
W Score	–	1.00	0.80	0.70	0.60	0.00

3.3.8. Hyperparameter Fidelity

3.3.9. Conclusion

All baselines attempt to implement PyNeRF with varying levels of sophistication. GPT-5 produces the cleanest modular code but lacks framework integration. DeepSeek R1 creates an ambitious architecture that remains largely unimplemented. Paper2Code achieves the most complete standalone implementation but fails due to a critical mathematical error in projected area computation. AutoP2C completely misunderstands the paper's core pyramid concept. Only NERFIFY produces immediately trainable code as a complete Nerfstudio plugin with correct mathematical formulations, achieving a 97% semantic completeness score. This stark contrast - 100% trainable rate for NERFIFY versus 0% for all baselines - demonstrates that effective paper-to-code translation for complex vision research requires deep domain specialization beyond what general-purpose systems can provide.

3.4. TensoRF: Tensorial Radiance Fields

3.4.1. Paper Overview

TensoRF [3] revolutionizes neural radiance fields by replacing computationally expensive MLPs with factorized 4D tensors using vector-matrix (VM) decomposition. The method decomposes the radiance field tensor $\mathcal{T} \in \mathbb{R}^{I \times J \times K \times P}$ into compact low-rank components, achieving 100× compression with superior quality. This explicit grid representation with tensor factorization demonstrates that properly structured explicit methods can outperform implicit neural representations in both reconstruction quality and training efficiency, achieving 30-minute training on a single GPU.

3.4.2. Implementation Overview

3.4.3. Novel Components

3.4.4. Quantitative Metrics

3.4.5. Component-by-Component Analysis

Component C1: Vector-Matrix Decomposition - All Baselines

The paper defines VM decomposition as: $\mathcal{T} = \sum_{r=1}^{R} \mathbf{v}_r^X \circ \mathbf{M}_r^{YZ} + \mathbf{v}_r^Y \circ \mathbf{M}_r^{XZ} + \mathbf{v}_r^Z \circ \mathbf{M}_r^{XY}$ where $\circ$ denotes outer product.

Table 40. TensoRF Implementation Summary Across All Baselines

Aspect	NERFIFY	GPT-5	DeepSeek R1	Paper2Code	AutoP2C
Lines of Code	467	298	412	856	234
File Organization	Plugin	Single	Single	Multi-file	Multi-file
VM Decomposition	✓	✓	Partial	×	×
Density Factorization	✓	✓	Partial	×	×
Appearance Matrix B	✓	✓	×	×	×
L1 Regularization	✓	Partial	Partial	×	×
TV Regularization	✓	×	×	×	×
Trainable	✓	×	×	×	×

Note: NERFIFY always produces trainable code as a complete Nerfstudio plugin

Table 41. Novel Components in TensoRF with Importance Weights

ID	Component	Weight w_i
C1	Vector-Matrix (VM) Decomposition	0.20
C2	Density Grid Factorization	0.15
C3	Appearance Grid Factorization	0.15
C4	Global Appearance Matrix B	0.10
C5	Factor-level Trilinear Interpolation	0.10
C6	L1 Sparsity Regularization	0.08
C7	Total Variation Regularization	0.07
C8	Coarse-to-Fine Upsampling	0.05
C9	SH/MLP Decoder	0.05
C10	Separate Density/Appearance Architecture	0.05

Table 42. TensoRF Implementation Coverage Metrics

Method	C	I	M	W	$\text{Score}_{\text{LLM}}$
NERFIFY (Ours)	1.00	0.00	0.00	0.95	0.98
GPT-5	0.70	0.10	0.20	0.75	0.72
DeepSeek R1	0.60	0.20	0.20	0.70	0.65
Paper2Code	0.20	0.30	0.50	0.30	0.12
AutoP2C	0.10	0.20	0.70	0.15	0.28

```
# \nerfify\ : Complete VM decomposition with all three modes
self.vx_sigma = nn.Parameter(0.1 * torch.randn(self.R_sigma, self.Nx))
self.vy_sigma = nn.Parameter(0.1 * torch.randn(self.R_sigma, self.Ny))
self.vz_sigma = nn.Parameter(0.1 * torch.randn(self.R_sigma, self.Nz))
self.MyZ_sigma = nn.Parameter(0.1 * torch.randn(self.R_sigma, self.Ny, self.Nz))
self.MxZ_sigma = nn.Parameter(0.1 * torch.randn(self.R_sigma, self.Nx, self.Nz))
self.MxY_sigma = nn.Parameter(0.1 * torch.randn(self.R_sigma, self.Nx, self.Ny))
Ax = (vx * MyZ).sum(dim=0)  # v_r^X o M_r^YZ
Ay = (vy * MxZ).sum(dim=0)  # v_r^Y o M_r^XZ
Az = (vz * MxY).sum(dim=0)  # v_r^Z o M_r^XY
sigma_raw = Ax + Ay + Az + self.density_bias
# CORRECT: Exact VM decomposition from paper Eq. 7
```

Listing 149. C1 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Correct math but non-standard PyTorch usage
class VMDecomp(nn.Module):
    def forward(self, xyz):
        x, y, z = xyz
        vx = F.linear(x, self.vectors[0].T)
```

```
        myz = F.bilinear(y, z, self.matrices[0])
        result += (vx * myz).sum(dim=-1)
# ISSUE: F.bilinear doesn't work this way in PyTorch
```

Listing 150. C1 Implementation: GPT-5 (Score: 0.9)

```
# DeepSeek R1: Structure present but flawed execution
class VMDecomposition(nn.Module):
    def forward(self, xyz):
        vx = F.linear(x, self.vectors[0].T)
        myz = F.bilinear(y, z, self.matrices[0])
        result += (vx * myz).sum(dim=-1)
# CRITICAL ERROR: Bilinear function misused
```

Listing 151. C1 Implementation: DeepSeek R1 (Score: 0.6)

```
# Paper2Code: Wrong architecture - implements NeRF
class TensoRFModel(nn.Module):
    def __init__(self, decomposition_type='VM'):
        self.density_mlp = self._build_mlp(64, 1)
        self.color_mlp = self._build_mlp(128, 2)
# CRITICAL ERROR: Uses MLPs, not tensor factorization
    !
```

Listing 152. C1 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: No model at all
class BlenderDataset:
    def load_images(self):
        self.images.append(image)
# MISSING: Entire TensoRF model absent
```

Listing 153. C1 Implementation: AutoP2C (Score: 0.0)

Analysis: VM decomposition is the core innovation. Only NERFIFY implements it perfectly. GPT-5 has correct math but PyTorch issues. DeepSeek R1 attempts but fails. Paper2Code/AutoP2C completely miss it.

Component C2: Density Grid Factorization - All Baselines

The density grid $\mathcal{G}_\sigma \in \mathbb{R}^{I\times J\times K}$ uses VM decomposition with $R_\sigma = 16$ components.

```
# \nerfify\ : Perfect density factorization
def get_density(self, ray_samples: RaySamples):
    x, y, z = self._flatten_positions(ray_samples)
    vx = _interp_vector(self.vx_sigma, x)
    vy = _interp_vector(self.vy_sigma, y)
    vz = _interp_vector(self.vz_sigma, z)
    MyZ = _interp_plane(self.MyZ_sigma, y, z)
    MxZ = _interp_plane(self.MxZ_sigma, x, z)
    MxY = _interp_plane(self.MxY_sigma, x, y)
    Ax = (vx * MyZ).sum(dim=0)
    Ay = (vy * MxZ).sum(dim=0)
    Az = (vz * MxY).sum(dim=0)
    sigma = trunc_exp(Ax + Ay + Az + self.
     density_bias)
# CORRECT: Density grid with VM decomposition and
     trunc_exp
```

Listing 154. C2 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Correct density factorization
def get_density(self, xyz):
    return F.softplus(self.density(xyz))
# CORRECT: Proper density with softplus activation
```

Listing 155. C2 Implementation: GPT-5 (Score: 0.9)

```
# DeepSeek R1: Mostly correct but missing details
def forward_density(self, pos):
    for r in range(self.R):
        density_vals = density_vals + term_x + term_y
      + term_z
    return density_vals
# ISSUE: Missing activation function
```

Listing 156. C2 Implementation: DeepSeek R1 (Score: 0.8)

```
# Paper2Code: Wrong - uses MLPs
def forward(self, x, d):
    encoded_x = self.encode_position(x)
    density = self.density_mlp(encoded_x)
# CRITICAL ERROR: MLP with positional encoding
```

Listing 157. C2 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: No density implementation
# MISSING: No density code found
```

Listing 158. C2 Implementation: AutoP2C (Score: 0.0)

Analysis: Density factorization correctly splits from appearance. NERFIFY and GPT-5 implement correctly, DeepSeek R1 partially, Paper2Code/AutoP2C fail completely.

Component C3: Appearance Grid Factorization - All Baselines

Appearance grid $\mathcal{G}_c \in \mathbb{R}^{I\times J\times K\times P}$ with $P = 27$ features and $R_c = 48$ components.

```
# \nerfify\ : Complete appearance factorization
self.vx_c = nn.Parameter(0.1 * torch.randn(self.R_c,
     self.Nx))
self.vy_c = nn.Parameter(0.1 * torch.randn(self.R_c,
     self.Ny))
self.vz_c = nn.Parameter(0.1 * torch.randn(self.R_c,
     self.Nz))
self.MyZ_c = nn.Parameter(0.1 * torch.randn(self.R_c,
      self.Ny, self.Nz))
self.MxZ_c = nn.Parameter(0.1 * torch.randn(self.R_c,
      self.Nx, self.Nz))
self.MxY_c = nn.Parameter(0.1 * torch.randn(self.R_c,
      self.Nx, self.Ny))
concat_contrib = torch.cat([Ax, Ay, Az], dim=0).
     transpose(0, 1)
app_feat = self.B(concat_contrib)  # [M, P=27]
# CORRECT: Separate appearance grid with feature
     dimension 27
```

Listing 159. C3 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Present but simplified
self.appearance = VMDecomposition(
    config.resolution,
    config.num_appearance_components)
# ISSUE: Missing feature dimension details
```

Listing 160. C3 Implementation: GPT-5 (Score: 0.7)

```
# DeepSeek R1: Partial implementation
def forward_appearance(self, pos):
    for r in range(self.R):
        feat_list.append(term_x)
        feat_list.append(term_y)
        feat_list.append(term_z)
# ERROR: Missing matrix B transformation
```

Listing 161. C3 Implementation: DeepSeek R1 (Score: 0.6)

```
# Paper2Code: No appearance grid
color = self.color_mlp(encoded_x)
# CRITICAL ERROR: Uses MLP, not factorized grid
```

Listing 162. C3 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: Missing appearance
# MISSING: No appearance implementation
```

Listing 163. C3 Implementation: AutoP2C (Score: 0.0)

Analysis: Appearance requires separate VM decomposition with 48 components and 27 features. Only NERFIFY gets all details right.

Component C4: Global Appearance Matrix B - All Baselines

Matrix $\mathbf{B} \in \mathbb{R}^{P \times 3R_c}$ maps concatenated features to final appearance.

```
# \nerfify\ : Perfect matrix B implementation
self.B = nn.Linear(3 * self.R_c, self.P, bias=False)
concat_contrib = torch.cat([Ax, Ay, Az], dim=0).
    transpose(0, 1)
app_feat = self.B(concat_contrib)  # [M, P]
# CORRECT: Matrix B as linear layer without bias
```

Listing 164. C4 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Correct matrix B
self.B = nn.Parameter(
    torch.randn(config.appearance_dim,
                3 * config.num_appearance_components)
    )
features = torch.matmul(vm_features, self.B.T)
# CORRECT: Proper matrix multiplication
```

Listing 165. C4 Implementation: GPT-5 (Score: 1.0)

```
# DeepSeek R1: Missing matrix B
# No global appearance matrix in code
# CRITICAL ERROR: Matrix B completely absent
```

Listing 166. C4 Implementation: DeepSeek R1 (Score: 0.0)

```
# Paper2Code: No matrix B
# CRITICAL ERROR: Architecture doesn't include matrix
    B
```

Listing 167. C4 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: No matrix B
# MISSING: No model implementation
```

Listing 168. C4 Implementation: AutoP2C (Score: 0.0)

Analysis: Matrix B is crucial for appearance. Only NERFIFY and GPT-5 implement it. Its absence severely impacts quality in other baselines.

Component C5: Factor-level Trilinear Interpolation - All Baselines

Efficient interpolation at factor level rather than dense grid: $\mathcal{A}_r^X(\mathbf{x}) = \mathbf{v}_r^X(x) \cdot \mathbf{M}_r^{YZ}(y, z)$

```
# \nerfify\ : Efficient factor-level interpolation
def _interp_vector(vec: Tensor, coord: Tensor) ->
    Tensor:
    scaled = coord.clamp(0.0, 1.0) * (N - 1)
    i0 = torch.floor(scaled).long().clamp(0, N - 1)
    i1 = (i0 + 1).clamp(0, N - 1)
    w = (scaled - i0.float()).clamp(0.0, 1.0)
    v0 = torch.gather(vec, 1, idx0)
    v1 = torch.gather(vec, 1, idx1)
    return (1.0 - w)[None, :] * v0 + w[None, :] * v1
# CORRECT: Efficient linear interpolation on factors
```

Listing 169. C5 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Correct interpolation approach
def interp(self, v, x):
    idx0 = torch.clamp(x.long(), 0, v.shape[1]-2)
    w = x - idx0.float()
    return (1-w)*v[:, idx0] + w*v[:, idx0+1]
# CORRECT: Factor interpolation implemented
```

Listing 170. C5 Implementation: GPT-5 (Score: 0.9)

```
# DeepSeek R1: Interpolation present but inefficient
def _interp1d(self, vec: torch.Tensor, coord: torch.
    Tensor):
    coord = coord.clamp(0, length - 1)
    idx0 = torch.floor(coord).long()
    t = coord - idx0.float()
# ISSUE: Suboptimal implementation
```

Listing 171. C5 Implementation: DeepSeek R1 (Score: 0.7)

```
# Paper2Code: No factor interpolation
# CRITICAL ERROR: Uses positional encoding instead
```

Listing 172. C5 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: No interpolation
# MISSING: No model code
```

Listing 173. C5 Implementation: AutoP2C (Score: 0.0)

Analysis: Factor-level interpolation is key to efficiency. NERFIFY and GPT-5 implement correctly, DeepSeek R1 partially.

Component C6: L1 Sparsity Regularization - All Baselines

L1 regularization on density factors with weight $\omega = 0.0004$.

```
# \nerfify\ : Complete L1 on all density factors
l1_terms = [
    self.vx_sigma.abs().mean(),
    self.vy_sigma.abs().mean(),
    self.vz_sigma.abs().mean(),
    self.MyZ_sigma.abs().mean(),
    self.MxZ_sigma.abs().mean(),
    self.MxY_sigma.abs().mean(),
]
losses["l1_density"] = sum(l1_terms) / len(l1_terms)
loss_dict["l1_density"] = self.config.l1_density_mult
    * regs["l1_density"]
# CORRECT: L1 with exact weight 4e-4
```

Listing 174. C6 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: L1 present but simplified
l1_loss = 0.0
for param in model.density.parameters():
    l1_loss += torch.abs(param).mean()
total_loss = render_loss + config.l1_weight * l1_loss
# ISSUE: Hardcoded weight
```

Listing 175. C6 Implementation: GPT-5 (Score: 0.7)

```
# DeepSeek R1: Wrong weight
def compute_l1_reg(self):
    l1_reg += torch.mean(torch.abs(param))
total_loss = mse_loss + 1e-4 * reg_loss
# ERROR: Uses 1e-4 instead of 4e-4
```

Listing 176. C6 Implementation: DeepSeek R1 (Score: 0.5)

```
# Paper2Code: No L1 regularization
# MISSING: No regularization terms
```

Listing 177. C6 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: No regularization
# MISSING: No training code
```

Listing 178. C6 Implementation: AutoP2C (Score: 0.0)

Analysis: L1 sparsity crucial for compact models. Only NERFIFY gets weight exact, others have errors or missing.

Component C7: Total Variation Regularization - All Baselines

TV regularization for smooth factors, especially important for few-shot scenarios.

```
# \nerfify\ : Complete TV regularization
def _tv_1d(self, vec: Tensor) -> Tensor:
    return (vec[:, 1:] - vec[:, :-1]).abs().mean()
def _tv_2d(self, mat: Tensor) -> Tensor:
    tv_u = (mat[:, 1:, :] - mat[:, :-1, :]).abs().
     mean()
    tv_v = (mat[:, :, 1:] - mat[:, :, :-1]).abs().
     mean()
    return 0.5 * (tv_u + tv_v)
tv_density = (self._tv_1d(self.vx_sigma) + self.
     _tv_1d(self.vy_sigma) +
                self._tv_1d(self.vz_sigma) + self.
     _tv_2d(self.MyZ_sigma) +
                self._tv_2d(self.MxZ_sigma) + self.
     _tv_2d(self.MxY_sigma)) / 6.0
# CORRECT: TV on both 1D vectors and 2D matrices
```

Listing 179. C7 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: No TV regularization
# MISSING: TV not implemented
```

Listing 180. C7 Implementation: GPT-5 (Score: 0.0)

```
# DeepSeek R1: No TV regularization
# MISSING: TV completely absent
```

Listing 181. C7 Implementation: DeepSeek R1 (Score: 0.0)

```
# Paper2Code: No TV
# MISSING: No regularization
```

Listing 182. C7 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: No TV
# MISSING: No model
```

Listing 183. C7 Implementation: AutoP2C (Score: 0.0)

Analysis: TV regularization critical for few-shot. Only NERFIFY implements it correctly with both 1D and 2D variants.

Component C8: Coarse-to-Fine Upsampling - All Baselines

Progressive resolution increase from 128^3 to 300^3 during training.

```
# \nerfify\ : Upsampling via training schedule
upsampling_schedule: [2000, 3000, 4000, 5500, 7000]
grid_resolution: (128, 128, 128)  # Initial
final_resolution: 300  # Target
# Training config handles progressive upsampling
# ISSUE: Not explicit upsampling, handled by
     scheduler
```

Listing 184. C8 Implementation: NERFIFY (Score: 0.8)

```
# GPT-5: No coarse-to-fine
# MISSING: No upsampling implementation
```

Listing 185. C8 Implementation: GPT-5 (Score: 0.0)

```
# DeepSeek R1: No upsampling
# MISSING: No coarse-to-fine schedule
```

Listing 186. C8 Implementation: DeepSeek R1 (Score: 0.0)

```
# Paper2Code: No upsampling
# MISSING: No progressive resolution
```

Listing 187. C8 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: No upsampling
# MISSING: No training pipeline
```

Listing 188. C8 Implementation: AutoP2C (Score: 0.0)

Analysis: Coarse-to-fine speeds convergence. NERFIFY implements via schedule, others completely miss it.

Component C9: SH/MLP Decoder - All Baselines

View-dependent color decoder using spherical harmonics or small MLP.

```
# \nerfify\ : Complete SH encoding + MLP decoder
self.direction_encoding = SHEncoding(levels=3,
     implementation="torch")
self.color_head = MLP(
    in_dim=self.P + self.direction_encoding.
     get_out_dim(),
    num_layers=color_mlp_layers,
    layer_width=color_mlp_width,
    out_dim=3,
    activation=nn.ReLU(),
    out_activation=nn.Sigmoid())
# CORRECT: SH encoding with 2-layer MLP decoder
```

Listing 189. C9 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: MLP decoder present
self.mlp = nn.Sequential(
    nn.Linear(3 * config.num_appearance_components,
     128),
    nn.ReLU(),
    nn.Linear(128, 3),
    nn.Sigmoid())
# ISSUE: Missing SH option
```

Listing 190. C9 Implementation: GPT-5 (Score: 0.7)

```
# DeepSeek R1: Partial decoder
self.mlp = nn.Sequential(
    nn.Linear(mlp_input_dim, mlp_hidden_size),
    nn.ReLU(),
    nn.Linear(mlp_hidden_size, 3))
# ERROR: Missing SH, incomplete MLP
```

Listing 191. C9 Implementation: DeepSeek R1 (Score: 0.6)

```
# Paper2Code: Wrong decoder type
self.color_mlp = self._build_mlp(128, 2)
# CRITICAL ERROR: Wrong architecture
```

Listing 192. C9 Implementation: Paper2Code (Score: 0.2)

```
# AutoP2C: Evaluation mentions MLP
# MISSING: No actual implementation
```

Listing 193. C9 Implementation: AutoP2C (Score: 0.1)

Analysis: Decoder maps appearance features to RGB. NERFIFY implements both SH and MLP options correctly.

Component C10: Separate Density/Appearance Architecture - All Baselines

Independent factorizations for density and appearance with different ranks.

```
# \nerfify\ : Completely separate density/appearance
self.density_factor = TensorFactorization(
    grid_shape=grid_shape,
    num_components={"density": 16},  # R_sigma = 16
    decomposition=decomposition,
    grid_type="density")
self.appearance_factor = TensorFactorization(
    grid_shape=grid_shape,
    num_components={"appearance": 48},  # R_c = 48
    decomposition=decomposition,
    grid_type="appearance")
# CORRECT: Separate factors with different ranks
```

Listing 194. C10 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Separate but less clear
self.density = VMDecomposition(
    config.resolution,
    config.num_density_components)
self.appearance = VMDecomposition(
    config.resolution,
    config.num_appearance_components)
# CORRECT: Separate architectures
```

Listing 195. C10 Implementation: GPT-5 (Score: 0.8)

```
# DeepSeek R1: Attempted separation
self.density_factor = TensorFactorization(
    grid_type="density")
self.appearance_factor = TensorFactorization(
    grid_type="appearance")
# ISSUE: Implementation details incomplete
```

Listing 196. C10 Implementation: DeepSeek R1 (Score: 0.7)

```
# Paper2Code: No separation
# CRITICAL ERROR: Single MLP for both
```

Listing 197. C10 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: No architecture
# MISSING: No model
```

Listing 198. C10 Implementation: AutoP2C (Score: 0.0)

Analysis: Separation allows different ranks for density (16) and appearance (48). Critical for efficiency/quality balance.

3.4.6. Scoring Analysis

Table 43. TensoRF Implementation Scores

Method	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	Weighted Avg$_{\text{LLM}}$
NERFIFY	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.8	1.0	1.0	0.98
GPT-5	0.9	0.9	0.7	1.0	0.9	0.7	0.0	0.0	0.7	0.8	0.72
DeepSeek R1	0.6	0.8	0.6	0.0	0.7	0.5	0.0	0.0	0.6	0.7	0.65
Paper2Code	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.0	0.12
AutoP2C	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.28

3.4.7. Why Baselines Fail Despite Component Scores

GPT-5 (Score: 0.72 components, 0% trainable)

- **Strengths**: Correct VM decomposition, matrix B implementation, density/appearance separation
- **Fatal Issues**:
  - No Nerfstudio integration - requires complete pipeline assembly
  - Missing TV regularization entirely (C7 = 0.0)
  - No coarse-to-fine upsampling (C8 = 0.0)
- **Result**: Mathematically sound but requires hours of engineering to make trainable

DeepSeek R1 (Score: 0.65 components, 0% trainable)

- **Strengths**: Understands tensor decomposition concept, attempts proper separation
- **Fatal Issues**:
  - PyTorch tensor operations cause runtime crashes (F.bilinear misuse)
  - Missing matrix B completely (C4 = 0.0)
  - No TV regularization (C7 = 0.0) causes training instability
- **Result**: Training crashes immediately on tensor shape mismatches

Paper2Code (Score: 0.12 components, 0% trainable)

- **Strengths**: Basic volume rendering loop exists
- **Fatal Issues**:
  - Complete architectural misunderstanding - implements vanilla NeRF
  - Zero correct components except partial decoder (C9 = 0.2)
  - Uses MLPs with positional encoding throughout
- **Result**: Trains as NeRF, not TensoRF - fundamentally wrong method

AutoP2C (Score: 0.28 components, 0% trainable)

- **Strengths**: Dataset loading code present
- **Fatal Issues**:
  - No model implementation whatsoever
  - Only evaluation stub mentions components (C9 = 0.1)
  - Missing entire TensoRF architecture
- **Result**: Code doesn't compile - no model to train

3.4.8. Hyperparameter Fidelity

Table 44. Hyperparameter Implementation Accuracy

Parameter	Paper	NERFIFY	GPT-5	R1	P2C	AutoP2C
Grid Resolution	128^3	✓	✓	×	–	–
Density Rank R_σ	16	✓	×	✓	–	–
Appearance Rank R_c	48	✓	✓	×	–	–
Feature Dim P	27	✓	✓	✓	–	–
L1 Weight	4e-4	✓	✓	×	–	–
TV Weight	Variable	✓	×	×	–	–
Learning Rate	0.02	×	✓	✓	×	–
Training Steps	30K	✓	×	×	–	–
MLP Width	128	✓	✓	✓	×	–
W Score	–	0.95	0.75	0.70	0.30	0.15

3.4.9. Conclusion

All baselines attempt to implement TensoRF with dramatically varying levels of success. GPT-5 achieves mathematically correct VM decomposition and matrix B but lacks critical TV regularization and framework integration. DeepSeek R1 understands the tensor factorization concept but fails catastrophically on PyTorch execution and missing matrix B. Paper2Code fundamentally misunderstands the method, implementing vanilla NeRF with MLPs and positional encoding instead of tensor factorization. AutoP2C provides only dataset loading code with no model implementation whatsoever. Despite GPT-5 achieving 72% component coverage and DeepSeek R1 reaching 65%, none produce trainable code due to missing Nerfstudio integration, incorrect tensor operations, or complete architectural misunderstandings. Only NERFIFY produces immediately trainable code as a complete Nerfstudio plugin, achieving 100% trainable rate versus 0% for all baselines, demonstrating the critical importance of domain-specific synthesis for complex computer vision research.

3.5. Tetra-NeRF: Representing Neural Radiance Fields Using Tetrahedra

3.5.1. Paper Overview

Tetra-NeRF [12] introduces an adaptive scene representation using Delaunay triangulation of point clouds, replacing uniform voxel grids with tetrahedra that naturally concentrate resolution near surfaces. The method achieves faster training and rendering by using barycentric interpolation within tetrahedra, eliminating the need for dense regular grids while maintaining quality comparable to state-of-the-art methods.

3.5.2. Implementation Overview

Table 45. Tetra-NeRF Implementation Summary Across All Baselines

Aspect	NERFIFY	GPT-5	DeepSeek R1	Paper2Code	AutoP2C
Lines of Code	412	189	267	845	621
File Organization	Plugin	Single	Single	Multi-file	Multi-file
Delaunay Triangulation	✓	✓	✓	Attempted	×
Barycentric Coords	✓	Simplified	✓	×	×
Feature Interpolation	✓	✓	✓	Partial	×
Volume Rendering	✓	∼	∼	×	×
Trainable	✓	×	×	×	×

Note: NERFIFY always produces trainable code as a complete Nerfstudio plugin

3.5.3. Novel Components

Table 46. Novel Components in Tetra-NeRF with Importance Weights

ID	Component	Weight w_i
C1	Delaunay triangulation for adaptive subdivision	0.20
C2	Barycentric coordinates using volume ratios (Eq. 2)	0.15
C3	Feature interpolation via barycentric weights	0.15
C4	Tetrahedral grid data structure	0.10
C5	Adaptive non-uniform resolution	0.10
C6	Hierarchical coarse-to-fine sampling	0.10
C7	Volume rendering with tetrahedral queries	0.08
C8	Tetrahedron volume computation	0.05
C9	Nerfstudio plugin integration	0.05
C10	RAdam optimizer with exponential decay	0.02

3.5.4. Quantitative Metrics

Table 47. Tetra-NeRF Implementation Coverage Metrics

Method	C	I	M	W	$\text{Score}_{\text{LLM}}$
NERFIFY (Ours)	1.00	0.00	0.00	1.00	1.00
GPT-5	0.50	0.25	0.25	0.60	0.58
DeepSeek R1	0.63	0.25	0.13	0.70	0.72
Paper2Code	0.13	0.25	0.63	0.20	0.22
AutoP2C	0.00	0.13	0.88	0.00	0.08

3.5.5. Component-by-Component Analysis

Component C1: Delaunay Triangulation - All Baselines

The paper specifies: "We represent the scene as a dense triangulation of the input point cloud, where the scene is a set of non-overlapping tetrahedra whose union is the convex hull of the original point cloud."

```
# \nerfify\ : Proper tetrahedral grid with simplex decomposition
class TetraNerfField(Field):
    def _tetrahedral_interpolate(self, positions_norm):
        # 6-simplex decomposition of unit cube
        scales = torch.tensor([self.nx-1, self.ny-1, self.nz-1])
```

```
        gpos = positions_norm * scales
        base = torch.floor(gpos).long()
        u = (gpos - base.to(gpos.dtype)).contiguous()
        # Sort u components for simplex determination
        u_sorted, perm = torch.sort(u, dim=1,
    descending=False)
        k1, k2, k3 = perm[:, 0], perm[:, 1], perm[:,
    2]
# CORRECT: Full tetrahedral decomposition with proper
     simplex
```

Listing 199. C1 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Basic Delaunay implementation
def build_tetrahedra(points):
    tri = Delaunay(points)
    verts = torch.tensor(points, dtype=torch.float32,
      device=device)
    tets = torch.tensor(tri.simplices, dtype=torch.
     long, device=device)
    return verts, tets
# CORRECT: Uses scipy.spatial.Delaunay correctly
```

Listing 200. C1 Implementation: GPT-5 (Score: 1.0)

```
# DeepSeek R1: Complete tetrahedral grid class
class TetrahedralGrid:
    def __init__(self, points: torch.Tensor):
        self.points = points.cpu().numpy()
        self.tetrahedra = Delaunay(self.points)
        self.features = nn.Parameter(
            torch.randn(len(points), 64) * 1e-4)
# CORRECT: Proper Delaunay with feature
     initialization
```

Listing 201. C1 Implementation: DeepSeek R1 (Score: 1.0)

```
# Paper2Code: Wrong approach
class DatasetLoader:
    def perform_delaunay_triangulation(self):
        delaunay = scipy.spatial.Delaunay(self.
     point_cloud)
        self.tetrahedra = delaunay.simplices
# ERROR: Not integrated with rendering pipeline
```

Listing 202. C1 Implementation: Paper2Code (Score: 0.2)

```
# AutoP2C: No triangulation at all
class ColmapDataset:
    def generate_point_cloud(self):
        self.point_cloud = np.array([point.xyz for
     point in
                                     reconstruction.
     points3D.values()])
# CRITICAL ERROR: Just loads points, no tetrahedra
```

Listing 203. C1 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY implements efficient simplex decomposition. GPT-5 and DeepSeek R1 correctly use Delaunay. Paper2Code attempts but fails to integrate. AutoP2C misses entirely.

Component C2: Barycentric Coordinates - All Baselines

The paper defines barycentric coordinates using volume ratios: $\lambda_i = \frac{V_i}{V_{\text{total}}}$ where V_i is the volume of sub-tetrahedron.

```
# \nerfify\ : Efficient simplex-based computation
def _tetrahedral_interpolate(self, positions_norm):
    u_sorted, perm = torch.sort(u, dim=1, descending=
     False)
    # Barycentric weights for simplex vertices
    w0 = 1.0 - u_sorted[:, 2]
    w1 = u_sorted[:, 2] - u_sorted[:, 1]
    w2 = u_sorted[:, 1] - u_sorted[:, 0]
    w3 = u_sorted[:, 0]
    # Weighted interpolation
    f = (f0 * w0.unsqueeze(-1) + f1 * w1.unsqueeze
     (-1) +
         f2 * w2.unsqueeze(-1) + f3 * w3.unsqueeze
     (-1))
# CORRECT: Efficient vectorized barycentric
     computation
```

Listing 204. C2 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Linear system approach
def barycentric_coords(p, tet_verts):
    V0, V1, V2, V3 = tet_verts
    M = torch.stack([V1-V0, V2-V0, V3-V0], dim=1)
    v = p - V0
    sol = torch.linalg.solve(M, v)
    l1, l2, l3 = sol
    l0 = 1 - l1 - l2 - l3
    return torch.stack([l0, l1, l2, l3])
# ISSUE: No volume validation, less robust
```

Listing 205. C2 Implementation: GPT-5 (Score: 0.4)

```
# DeepSeek R1: Volume-based as in paper
def tetrahedron_volume(self, a, b, c, d):
    return torch.abs(torch.dot(a - d,
                     torch.cross(b - d, c - d))) /
     6.0
def get_barycentric_coords(self, query_points,
     tetra_indices):
    V_total = self.tetrahedron_volume(v0, v1, v2, v3)
    V_x234 = self.tetrahedron_volume(query_points[i],
      v1, v2, v3)
    V_1x34 = self.tetrahedron_volume(v0, query_points
     [i], v2, v3)
    barycentric_coords[i] = torch.tensor([V_x234/
     V_total, ...])
# CORRECT: Exact paper formula with volume ratios
```

Listing 206. C2 Implementation: DeepSeek R1 (Score: 1.0)

```
# Paper2Code: Missing implementation
def compute_barycentrics(self, query_point,
     simplex_index):
    transform_matrix = self.delaunay.transform[
     simplex_index]
    T = transform_matrix[:ndim, :]
    offset = transform_matrix[ndim, :]
    # Function incomplete, never returns values
# CRITICAL ERROR: Never computes actual coordinates
```

Listing 207. C2 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: No barycentric coordinates
# Uses completely different approach
# MISSING: Core mathematical concept entirely
```

Listing 208. C2 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY uses efficient simplex computation. DeepSeek R1 implements exact volume method. GPT-5 uses valid but less robust approach. Paper2Code and AutoP2C fail completely.

Component C3: Feature Interpolation - All Baselines

The paper specifies interpolating features: $\mathbf{f}(\mathbf{q}) = \sum_{i=0}^{3} \lambda_i \mathbf{f}_i$

```python
# \nerfify\ : Efficient vectorized interpolation
def _gather_grid(self, idx):
    ix, iy, iz = idx.unbind(-1)
    grid_flat = self.features.view(-1, self.vertex_feature_dim)
    lin = ix * (self.ny * self.nz) + iy * self.nz + iz
    return grid_flat[lin]
# Gather and interpolate
f0 = self._gather_grid(v0)
f1 = self._gather_grid(v1)
f = (f0 * w0.unsqueeze(-1) + f1 * w1.unsqueeze(-1) +
     f2 * w2.unsqueeze(-1) + f3 * w3.unsqueeze(-1))
# CORRECT: Vectorized for efficiency
```

Listing 209. C3 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Correct but inefficient
def interpolate_feature(p, verts, feats, tets):
    centroids = verts[tets].mean(dim=1)
    idx = torch.argmin(torch.norm(centroids - p, dim=1))
    tet = tets[idx]
    lambdas = barycentric_coords(p, verts[tet])
    f = (lambdas[:, None] * feats[tet]).sum(0)
    return f
# ISSUE: O(n) search instead of O(log n)
```

Listing 210. C3 Implementation: GPT-5 (Score: 0.8)

```python
# DeepSeek R1: Correct interpolation
def interpolate_features(self, query_points):
    for i in range(batch_size):
        if tetra_indices[i] >= 0:
            tetra_vertices = self.tetrahedra.simplices[tetra_indices[i]]
            vertex_features = self.features[tetra_vertices]
            interpolated_features[i] = torch.sum(
                barycentric[i].unsqueeze(1) * vertex_features, dim=0)
# CORRECT: Proper weighted sum
```

Listing 211. C3 Implementation: DeepSeek R1 (Score: 1.0)

```python
# Paper2Code: Wrong method
def batch_barycentric_interpolation(self, query_points):
    for i in range(query_points.shape[0]):
        try:
            feat = self.barycentric_interpolation(pt, device=device)
        except ValueError:
            feat = torch.zeros(self.vertex_features.shape[1])
# ERROR: Returns zeros instead of interpolating
```

Listing 212. C3 Implementation: Paper2Code (Score: 0.3)

```python
# AutoP2C: No feature interpolation
# MISSING: Core rendering component
```

Listing 213. C3 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY provides vectorized implementation. GPT-5 correct but inefficient. DeepSeek R1 correct with loops. Paper2Code fails. AutoP2C missing.

Component C4: Tetrahedral Grid Structure - All Baselines

The paper requires a data structure to manage vertices, features, and connectivity.

```python
# \nerfify\ : Complete TetraNerfField class
class TetraNerfField(Field):
    def __init__(self, aabb, grid_resolution=32,
                 vertex_feature_dim=64,
                 bottleneck_dim=32):
        super().__init__()
        self.register_buffer("aabb", aabb)
        self.nx = self.ny = self.nz = grid_resolution
        self.features = nn.Parameter(torch.empty(
            self.nx, self.ny, self.nz,
            vertex_feature_dim))
        nn.init.uniform_(self.features, a=-1e-4, b=1e-4)
        self.base_mlp = MLP(in_dim=vertex_feature_dim,
                            num_layers=3, layer_width=128)
# CORRECT: Complete field with Nerfstudio integration
```

Listing 214. C4 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Functions only, no proper class
def build_tetrahedra(points):
    tri = Delaunay(points)
    verts = torch.tensor(points)
    tets = torch.tensor(tri.simplices)
def interpolate_feature(p, verts, feats, tets):
    # Separate functions
# ISSUE: No unified data structure
```

Listing 215. C4 Implementation: GPT-5 (Score: 0.5)

```python
# DeepSeek R1: Has structure but incomplete
class TetrahedralGrid:
    def __init__(self, points):
        self.points = points.cpu().numpy()
        self.tetrahedra = Delaunay(self.points)
        self.features = nn.Parameter(
            torch.randn(len(points), 64) * 1e-4)
    def find_tetrahedron(self, query_points):
        query_np = query_points.cpu().numpy()
        tetra_indices = self.tetrahedra.find_simplex(query_np)
# CORRECT: Good structure but missing integration
```

Listing 216. C4 Implementation: DeepSeek R1 (Score: 0.8)

```python
# Paper2Code: Wrong structure
class TetrahedraField(nn.Module):
    def __init__(self, point_cloud,
                 vertex_feature_dim=64):
        self.points = point_cloud
        self.delaunay = None  # Never initialized
# CRITICAL ERROR: Empty module
```

Listing 217. C4 Implementation: Paper2Code (Score: 0.0)

```python
# AutoP2C: No grid structure
# MISSING: Core data structure
```

Listing 218. C4 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY provides complete structure. GPT-5 lacks organization. DeepSeek R1 has structure but incomplete. Paper2Code and AutoP2C fail.

Component C5: Adaptive Resolution - All Baselines

The paper achieves adaptive resolution through Delaunay triangulation concentrating tetrahedra near surfaces.

```python
# \nerfify\ : Implicit adaptive via simplex decomposition
def _tetrahedral_interpolate(self, positions_norm):
    # Scale to grid coordinates
    scales = torch.tensor([self.nx-1, self.ny-1, self.nz-1])
    gpos = positions_norm * scales
    base = torch.floor(gpos).long()
    # Smaller tetrahedra near surfaces due to decomposition
    base[:, 0].clamp_(0, self.nx - 2)
    base[:, 1].clamp_(0, self.ny - 2)
    base[:, 2].clamp_(0, self.nz - 2)
# CORRECT: Achieves adaptivity through decomposition
```

Listing 219. C5 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: No adaptive handling
t_vals = torch.linspace(0.0, 1.0, n_samples)
# Uniform sampling without adaptation
# MISSING: Adaptive resolution concept
```

Listing 220. C5 Implementation: GPT-5 (Score: 0.2)

```python
# DeepSeek R1: Structure supports but not implemented
self.tetrahedra = Delaunay(self.points)
# Natural adaptivity from Delaunay but not utilized
# ISSUE: Has potential but not exploited
```

Listing 221. C5 Implementation: DeepSeek R1 (Score: 0.5)

```python
# Paper2Code: Opposite of adaptive
# Uses uniform grid
# CRITICAL ERROR: Misses key innovation
```

Listing 222. C5 Implementation: Paper2Code (Score: 0.0)

```python
# AutoP2C: Random points
random_points = np.random.normal(
    loc=self.point_cloud.mean(axis=0),
    scale=self.point_cloud.std(axis=0))
# MISSING: Any spatial structure
```

Listing 223. C5 Implementation: AutoP2C (Score: 0.0)

Analysis: Only NERFIFY achieves adaptive resolution. DeepSeek R1 has structure but doesn't exploit. Others miss innovation.

Component C6: Hierarchical Sampling - All Baselines

The paper uses coarse-to-fine sampling with importance weighting.

```python
# \nerfify\ : Inherits from Nerfacto's samplers
class TetraNerfModel(NerfactoModel):
    def populate_modules(self):
        Model.populate_modules(self)
        super().populate_modules()  # Gets proposal networks
        # Uses Nerfacto's production hierarchical sampling
        self.field = TetraNerfField(aabb=self.scene_box.aabb)
# CORRECT: Production-quality hierarchical sampling
```

Listing 224. C6 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Basic implementation
def hierarchical_sampling(ray_o, ray_d, model, n_coarse=32, n_fine=32):
    t_coarse = torch.linspace(0, 1, n_coarse)
    # Fine sampling proportional to weights
    cdf = torch.cumsum(w_norm.squeeze(), 0)
    u = torch.rand(n_fine, device=device)
    t_fine = torch.interp(u, cdf, t_coarse)
# ISSUE: Simplified, missing details
```

Listing 225. C6 Implementation: GPT-5 (Score: 0.6)

```python
# DeepSeek R1: Complete but inefficient
def volume_render(self, rays_o, rays_d, near, far):
    # Coarse sampling
    t_vals_coarse = torch.linspace(near, far, self.config.num_samples_coarse)
    weights = self.compute_weights(density_coarse, t_vals_coarse)
    # Fine sampling
    t_vals_fine = self.sample_fine(t_vals_coarse, weights)
# CORRECT: Right idea but loop-based
```

Listing 226. C6 Implementation: DeepSeek R1 (Score: 0.8)

```python
# Paper2Code: Only mentions
def stratified_sampling(self, nears, fars, num_samples):
    t_bins = torch.linspace(0.0, 1.0, steps=num_samples + 1)
    # Only coarse, no fine
# ERROR: Missing importance sampling
```

Listing 227. C6 Implementation: Paper2Code (Score: 0.2)

```python
# AutoP2C: Random sampling
def generate_rays(self):
    for pose in self.camera_poses:
        rays = self._generate_rays_for_pose(pose)
# MISSING: Hierarchical concept
```

Listing 228. C6 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY leverages production samplers. GPT-5 and DeepSeek R1 attempt with varying success. Paper2Code and AutoP2C fail.

Component C7: Volume Rendering - All Baselines

The paper requires volume rendering with tetrahedral field queries.

```python
# \nerfify\ : Complete volume rendering
def get_outputs(self, ray_samples, density_embedding=None):
    dirs = get_normalized_directions(ray_samples.frustums.directions)
    dirs_flat = dirs.view(-1, 3)
    denc = self.direction_encoding(dirs_flat)
    h = torch.cat([denc, density_embedding.view(-1, self.bottleneck_dim)], dim=-1)
    rgb = self.color_head(h).view(*dirs.shape[:-1], -1).to(dirs)
    outputs[FieldHeadNames.RGB] = rgb
    return outputs
# CORRECT: Full integration with Nerfstudio rendering
```

Listing 229. C7 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Basic volume rendering
def volume_render(ray_o, ray_d, model, verts, feats, tets, n_samples=64):
    t_vals = torch.linspace(0.0, 1.0, n_samples)
    pts = ray_o[None, :] + ray_d[None, :] * t_vals[:, None]
    weights = sigmas * T * deltas[:, None]
    rgb = (weights * colors).sum(0)
```

```
    return rgb
# CORRECT: Basic but functional
```

Listing 230. C7 Implementation: GPT-5 (Score: 0.7)

```
# DeepSeek R1: Volume rendering with issues
def compute_weights(self, density, t_vals):
    delta = t_vals[..., 1:] - t_vals[..., :-1]
    alpha = 1 - torch.exp(-density * delta)
    transmittance = torch.cumprod(1 - alpha + 1e-10,
     dim=-1)
    weights = alpha * transmittance
# ISSUE: Incorrect transmittance computation
```

Listing 231. C7 Implementation: DeepSeek R1 (Score: 0.6)

```
# Paper2Code: Wrong rendering
def volume_rendering(self, t_vals, sigma, rgb):
    # Missing proper implementation
# CRITICAL ERROR: No actual rendering
```

Listing 232. C7 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: No volume rendering
# MISSING: Core rendering pipeline
```

Listing 233. C7 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY has full integration. GPT-5 basic but functional. DeepSeek R1 has errors. Paper2Code and AutoP2C missing.

Component C8: Tetrahedron Volume - All Baselines

The paper computes tetrahedron volume: $V = \frac{1}{6}|(\mathbf{a} - \mathbf{d}) \cdot ((\mathbf{b} - \mathbf{d}) \times (\mathbf{c} - \mathbf{d}))|$

```
# \nerfify\ : Implicit in simplex decomposition
# Volume computation implicit in barycentric weights
w0 = 1.0 - u_sorted[:, 2]  # Volume ratio
w1 = u_sorted[:, 2] - u_sorted[:, 1]
w2 = u_sorted[:, 1] - u_sorted[:, 0]
w3 = u_sorted[:, 0]
# CORRECT: Efficient volume-based weights
```

Listing 234. C8 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: No explicit volume computation
# Uses linear algebra instead
# ISSUE: Missing volume formula
```

Listing 235. C8 Implementation: GPT-5 (Score: 0.3)

```
# DeepSeek R1: Exact formula
def tetrahedron_volume(self, a, b, c, d):
    return torch.abs(torch.dot(a - d,
                     torch.cross(b - d, c - d))) /
     6.0
# CORRECT: Exact paper formula
```

Listing 236. C8 Implementation: DeepSeek R1 (Score: 1.0)

```
# Paper2Code: No volume computation
# MISSING: Mathematical component
```

Listing 237. C8 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: Missing entirely
# MISSING: Core formula
```

Listing 238. C8 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY implicit in weights. DeepSeek R1 exact formula. GPT-5 missing. Paper2Code and AutoP2C absent.

Component C9: Nerfstudio Integration - All Baselines

The paper implementation requires Nerfstudio plugin structure for training.

```
# \nerfify\ : Complete Nerfstudio plugin
tetra_nerf = MethodSpecification(
    config=TrainerConfig(
        method_name="tetra-nerf",
        pipeline=TetraNerfPipelineConfig(
            datamanager=TetraNerfDataManagerConfig(),
            model=TetraNerfModelConfig()),
        optimizers={
            "fields": {"optimizer":
     RAdamOptimizerConfig(lr=1e-3)}}),
    description="Tetra-NeRF: Nerfacto-based with
     tetrahedral field")
# CORRECT: Complete plugin specification
```

Listing 239. C9 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: No Nerfstudio integration
if __name__ == "__main__":
    # Standalone demo only
    model, verts, feats, tets = train_tetra_nerf(pts,
      rays_o, rays_d)
# MISSING: Plugin structure
```

Listing 240. C9 Implementation: GPT-5 (Score: 0.0)

```
# DeepSeek R1: No framework integration
class TetraNeRFTrainer:
    def __init__(self, model, config):
        self.model = model
        self.optimizer = torch.optim.Adam(model.
     parameters())
# MISSING: Nerfstudio plugin
```

Listing 241. C9 Implementation: DeepSeek R1 (Score: 0.0)

```
# Paper2Code: Custom training only
class Trainer:
    def train(self):
        for iter_num in range(1, self.iterations + 1)
     :
# MISSING: Framework integration
```

Listing 242. C9 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: No framework awareness
# MISSING: Plugin structure
```

Listing 243. C9 Implementation: AutoP2C (Score: 0.0)

Analysis: Only NERFIFY has Nerfstudio integration. All baselines lack plugin structure, making them untrainable in framework.

Component C10: RAdam Optimizer - All Baselines

The paper specifies RAdam optimizer with exponential decay from 1e-3 to 1e-4.

```
# \nerfify\ : Exact paper specification
"fields": {
    "optimizer": RAdamOptimizerConfig(lr=1e-3, eps=1e
     -8),
    "scheduler": ExponentialDecaySchedulerConfig(
        lr_final=1e-4, max_steps=300000),
```

```
}
# CORRECT: RAdam with proper decay
```

Listing 244. C10 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Correct optimizer
self.optimizer = torch.optim.RAdam(
    list(model.parameters()) + [feats], lr=1e-3)
self.scheduler = torch.optim.lr_scheduler.
    ExponentialLR(
    self.optimizer, gamma=0.999)
# CORRECT: RAdam with decay
```

Listing 245. C10 Implementation: GPT-5 (Score: 1.0)

```
# DeepSeek R1: Wrong optimizer
self.optimizer = torch.optim.Adam(model.parameters(),
     lr=config.learning_rate)
self.scheduler = torch.optim.lr_scheduler.
    ExponentialLR(
    self.optimizer, gamma=0.1)
# ERROR: Uses Adam instead of RAdam
```

Listing 246. C10 Implementation: DeepSeek R1 (Score: 0.8)

```
# Paper2Code: Correct optimizer setup
self.optimizer = torch.optim.RAdam(self.model.
    parameters(), lr=self.lr_initial)
def adjust_learning_rate(self, current_iter):
    decay_factor = (self.lr_final / self.lr_initial)
    **
                   (current_iter / self.
    lr_decay_steps)
# CORRECT: RAdam with exponential decay
```

Listing 247. C10 Implementation: Paper2Code (Score: 1.0)

```
# AutoP2C: Mentions RAdam
# Paper uses RAdam with exponential LR decay
# But implementation missing
# ISSUE: No actual optimizer code
```

Listing 248. C10 Implementation: AutoP2C (Score: 0.8)

Analysis: NERFIFY , GPT-5, and Paper2Code correctly implement RAdam. DeepSeek R1 uses Adam. AutoP2C mentions but doesn't implement.

3.5.6. Scoring Analysis

Table 48. Tetra-NeRF Implementation Scores

Method	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	Weighted Avg$_{LLM}$
NERFIFY	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.00
GPT-5	1.0	0.4	0.8	0.5	0.2	0.6	0.7	0.3	0.0	1.0	0.58
DeepSeek R1	1.0	1.0	1.0	0.8	0.5	0.8	0.6	1.0	0.0	0.8	0.72
Paper2Code	0.2	0.0	0.3	0.0	0.0	0.2	0.0	0.0	0.0	1.0	0.22
AutoP2C	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.8	0.08

3.5.7. Why Baselines Fail Despite Component Scores

GPT-5 (Score: 0.58 components, 0% trainable)

- **Strengths**: Implements core tetrahedral concepts correctly, proper Delaunay triangulation
- **Fatal Issues**:
  - No Nerfstudio integration whatsoever
  - Inefficient O(n) tetrahedron search instead of O(log n)
  - Missing training loop and data loading
- **Result**: Code runs in isolation but cannot be trained within Nerfstudio framework

DeepSeek R1 (Score: 0.72 components, 0% trainable)

- **Strengths**: Mathematically correct volume-based barycentric coordinates, proper feature interpolation
- **Fatal Issues**:
  - No Nerfstudio plugin structure
  - Loop-based processing causes memory overflow on real datasets
  - Incomplete volume rendering implementation
- **Result**: Theoretically correct but practically unusable for training

Paper2Code (Score: 0.22 components, 0% trainable)

- **Strengths**: Attempts multi-file organization, includes optimizer setup
- **Fatal Issues**:
  - Fundamental misunderstanding: implements voxel grid instead of tetrahedra
  - Missing barycentric coordinate computation entirely
  - PyTorch errors including device mismatches
- **Result**: Implements a completely different method than the paper specifies

AutoP2C (Score: 0.08 components, 0% trainable)

- **Strengths**: Includes COLMAP data loading
- **Fatal Issues**:
  - Missing all core tetrahedral concepts
  - No Delaunay triangulation or barycentric coordinates
  - Non-functional Python with import errors
- **Result**: Non-functional code that fails at import stage

3.5.8. Hyperparameter Fidelity

Table 49. Hyperparameter Implementation Accuracy

Parameter	Paper	NERFIFY	GPT-5	R1	P2C	AutoP2C
Feature dimension	64	✓	✓	✓	×	×
Hidden dimension	128	✓	✓	✓	×	×
MLP layers	3	✓	✓	✓	×	×
Coarse samples	64	✓	×	✓	×	–
Fine samples	128	✓	×	✓	×	–
Learning rate	1e-3	✓	✓	✓	×	–
Optimizer	RAdam	✓	×	×	×	–
W Score	–	1.00	0.60	0.70	0.20	0.00

3.5.9. Conclusion

All baselines attempt to implement Tetra-NeRF with varying levels of sophistication. GPT-5 and DeepSeek R1 demonstrate understanding of core tetrahedral concepts, correctly implementing Delaunay triangulation and barycentric interpolation. Paper2Code fundamentally misunderstands the paper, implementing a voxel-based approach instead of tetrahedra. AutoP2C fails to implement any meaningful components beyond basic data loading. Despite GPT-5 achieving 58% and DeepSeek R1 achieving 72% component coverage,

neither produces trainable code due to missing Nerfstudio integration, inefficient algorithms, and incomplete training pipelines. Only NERFIFY produces immediately trainable code as a complete Nerfstudio plugin with 100% component fidelity. The 0% trainable rate for all baselines versus 100% for NERFIFY demonstrates the critical importance of domain-specific understanding and framework integration in neural rendering research. "'

3.6. E-NeRF: Neural Radiance Fields from a Moving Event Camera

3.6.1. Paper Overview

E-NeRF introduces a groundbreaking approach to neural radiance field reconstruction using event cameras, which capture pixel-level brightness changes rather than absolute intensity. The method replaces traditional RGB photometric losses with event-generation constraints, enabling NeRF reconstruction from asynchronous event streams. This is particularly valuable for scenarios with fast motion, low light, or high dynamic range where conventional cameras fail. The paper demonstrates that event supervision alone can produce high-quality 3D reconstructions, with optional RGB frame integration for enhanced color fidelity.

3.6.2. Implementation Overview

Table 50. E-NeRF Implementation Summary Across All Baselines

Aspect	NERFIFY	GPT-5	DeepSeek R1	Paper2Code	AutoP2C
Lines of Code	487	245	389	1650	156
File Organization	Plugin	Single	Single	Multi-file	Multi-file
Event Loss (Eq. 3)	✓	✓	✓	Partial	×
Linlog Mapping	✓	✓	✓	✓	×
No-Event Loss	✓	Partial	✓	×	×
Framework Integration	✓	×	×	×	×
Trainable	✓	×	×	×	×

Note: NERFIFY always produces trainable code as a complete Nerfstudio plugin

3.6.3. Novel Components

Table 51. Novel Components in E-NeRF with Importance Weights

ID	Component	Weight w_i
C1	Event-based loss function (Eq. 3)	0.25
C2	Linlog brightness mapping (Eq. 2)	0.20
C3	Normalized event loss (Eq. 4)	0.15
C4	No-event loss (Eq. 5)	0.10
C5	Event + RGB combined loss (Eq. 6)	0.10
C6	Event pair sampling strategy	0.08
C7	Camera pose interpolation (slerp/cubic)	0.07
C8	Hash-based encoding (Instant-NGP)	0.05

Table 52. E-NeRF Implementation Coverage Metrics

Method	C	I	M	W	$\text{Score}_{\text{LLM}}$
NERFIFY (Ours)	1.00	0.00	0.00	0.95	1.00
GPT-5	0.50	0.25	0.25	0.75	0.60
DeepSeek R1	0.63	0.25	0.13	0.80	0.72
Paper2Code	0.38	0.25	0.38	0.60	0.48
AutoP2C	0.00	0.13	0.88	0.00	0.05

3.6.4. Quantitative Metrics

3.6.5. Component-by-Component Analysis

Component C1: Event-based Loss Function - All Baselines

The paper's core innovation is the event loss enforcing brightness change constraints: $\mathcal{L}_{\text{evs}}(\Theta) = \|\Delta\hat{\mathbf{L}}(\Theta) - \Delta\mathbf{L}(\Theta)\|_2^2$ where $\Delta L_k = p_k C$ with contrast threshold $C = 0.2$.

```python
# \nerfify\ : Full integration with event batch handling
def _event_losses(self, batch):
    rb_t0 = self._bundle_from_batch_prefix(batch, "event_t0")
    rb_t1 = self._bundle_from_batch_prefix(batch, "event_t1")

    I0 = self._render_intensity(rb_t0)
    I1 = self._render_intensity(rb_t1)
    L0 = _linlog(I0, B)
    L1 = _linlog(I1, B)
    delta_hat = (L1 - L0).reshape(-1)

    delta_gt = (C * batch["event_polarity"]).reshape(-1)
    ev_loss = F.mse_loss(delta_hat, delta_gt)
    # CORRECT: Proper batch structure and loss computation
```

Listing 249. C1 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Mathematically correct but no framework integration
def event_loss(deltaL_hat, deltaL_true, normalized=False):
    if normalized:
        deltaL_hat = deltaL_hat / (torch.norm(deltaL_hat) + 1e-8)
        deltaL_true = deltaL_true / (torch.norm(deltaL_true) + 1e-8)
    return F.mse_loss(deltaL_hat, deltaL_true)
# ISSUE: No batch handling, standalone function
```

Listing 250. C1 Implementation: GPT-5 (Score: 0.8)

```python
# DeepSeek R1: Comprehensive but missing integration
def generate_event_loss(self, pred_log_brightness1,
                        pred_log_brightness2,
                        measured_polarity, C=0.2):
    pred_delta_L = pred_log_brightness2 - pred_log_brightness1
    target_delta_L = measured_polarity * C
    event_loss = F.mse_loss(pred_delta_L, target_delta_L)
    return event_loss
# CORRECT: Formula implementation
# MISSING: Training loop integration
```

Listing 251. C1 Implementation: DeepSeek R1 (Score: 0.9)

```python
# Paper2Code: Basic structure without proper batching
if self.event_loss_type == "standard":
    event_loss = torch.mean((predicted_diff - event_target_diff_tensor) ** 2)
elif self.event_loss_type == "normalized":
    event_loss = Utils.normalized_event_loss(predicted_diff, event_target_diff_tensor)
# ERROR: Missing batch structure, incomplete integration
```

Listing 252. C1 Implementation: Paper2Code (Score: 0.5)

```python
# AutoP2C: No implementation
# File only contains dataset placeholders
# CRITICAL ERROR: Event loss not implemented at all
```

Listing 253. C1 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY correctly implements the event loss with full Nerfstudio integration. GPT-5 and DeepSeek R1 have mathematically correct formulas but lack framework integration. Paper2Code attempts the loss but misses critical batch handling. AutoP2C completely lacks any loss implementation.

Component C2: Linlog Brightness Mapping - All Baselines

The paper specifies: $L(\mathbf{u}) = I(\mathbf{u}) \cdot \ln(B)/B$ if $I < B$, else $\ln(I)$ with $B = 20$.

```python
# \nerfify\ : Exact paper formula with proper epsilon handling
def _linlog(intensity: Tensor, B: float) -> Tensor:
    eps = 1e-6
    lnB_over_B = float(torch.log(torch.tensor(B)).item() / B)
    I = torch.clamp(intensity, min=eps)
    linear = I * lnB_over_B
    logv = torch.log(I)
    return torch.where(I < B, linear, logv)
# CORRECT: Exact implementation from paper
```

Listing 254. C2 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Correct implementation with masking
def linlog(I, B=20.0):
    mask = (I < B).float()
    return mask * (I * torch.log(torch.tensor(B)) / B) + \
           (1 - mask) * torch.log(I + 1e-6)
# CORRECT: Proper branching logic
```

Listing 255. C2 Implementation: GPT-5 (Score: 1.0)

```python
# DeepSeek R1: Correct with explicit masking
def linlog(self, I):
    linear_mask = I < self.B
    log_mask = ~linear_mask
    L = torch.zeros_like(I)
    L[linear_mask] = I[linear_mask] * torch.log(torch.tensor(self.B)) / self.B
    L[log_mask] = torch.log(I[log_mask])
    return L
# CORRECT: Accurate implementation
```

Listing 256. C2 Implementation: DeepSeek R1 (Score: 1.0)

```python
# Paper2Code: Verbose but correct
def lin_log_mapping(self, intensity: torch.Tensor) -> torch.Tensor:
    B_tensor = torch.tensor(B, device=intensity.device)
    return torch.where(
        intensity < B_tensor,
        intensity * torch.log(B_tensor) / B_tensor,
        torch.log(intensity + 1e-6)
    )
# CORRECT: Proper implementation
```

Listing 257. C2 Implementation: Paper2Code (Score: 1.0)

```python
# AutoP2C: Not implemented
# CRITICAL ERROR: No linlog mapping found in codebase
```

Listing 258. C2 Implementation: AutoP2C (Score: 0.0)

Analysis: All baselines except AutoP2C correctly implement linlog mapping. This relatively straightforward component shows that baselines can handle individual mathematical formulas but struggle with system integration.

Component C3: Normalized Event Loss - All Baselines

For unknown contrast threshold: $\mathcal{L}_{\text{norm}} = \left\| \frac{\Delta\hat{\mathbf{L}}}{\|\Delta\hat{\mathbf{L}}\|_2} - \frac{\Delta\mathbf{L}}{\|\Delta\mathbf{L}\|_2} \right\|_2^2$

```python
# \nerfify\ : Config-switchable normalized variant
if self.config.use_normalized_event_loss:
    eps = 1e-8
    nhat = delta_hat / (torch.linalg.norm(delta_hat) + eps)
    ngt = delta_gt / (torch.linalg.norm(delta_gt) + eps)
    ev_loss = F.mse_loss(nhat, ngt)
else:
    ev_loss = F.mse_loss(delta_hat, delta_gt)
# CORRECT: Proper L2 normalization before MSE
```

Listing 259. C3 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Has normalized variant as parameter
def event_loss(deltaL_hat, deltaL_true, normalized=False):
    if normalized:
        deltaL_hat = deltaL_hat / (torch.norm(deltaL_hat) + 1e-8)
        deltaL_true = deltaL_true / (torch.norm(deltaL_true) + 1e-8)
    return F.mse_loss(deltaL_hat, deltaL_true)
# CORRECT: But not integrated with training
```

Listing 260. C3 Implementation: GPT-5 (Score: 0.8)

```python
# DeepSeek R1: Separate function for normalized loss
def normalized_event_loss(self, pred_delta_L, target_delta_L):
    pred_norm = pred_delta_L / (torch.norm(pred_delta_L, dim=-1, keepdim=True) + 1e-8)
    target_norm = target_delta_L / (torch.norm(target_delta_L, dim=-1, keepdim=True) + 1e-8)
    return F.mse_loss(pred_norm, target_norm)
# CORRECT: Proper implementation
```

Listing 261. C3 Implementation: DeepSeek R1 (Score: 1.0)

```
# Paper2Code: No normalized variant
# Only has standard event loss
# CRITICAL ERROR: Missing normalized loss entirely
```

Listing 262. C3 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: Not implemented
# CRITICAL ERROR: No loss functions at all
```

Listing 263. C3 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY , GPT-5, and DeepSeek R1 correctly implement the normalized variant crucial for real-world data. Paper2Code and AutoP2C completely miss this component.

Component C4: No-Event Loss - All Baselines

Penalizes brightness changes in areas without events: $\mathcal{L}_{\text{noevs}} = \sum_k \text{relu}(|\hat{L}_k - \hat{L}_{k-1}| - C)$

```
# \nerfify\ : Complete no-event handling
rb_n0 = self._bundle_from_batch_prefix(batch, "noevent_t0")
rb_n1 = self._bundle_from_batch_prefix(batch, "noevent_t1")
if rb_n0 is not None and rb_n1 is not None:
    I0 = self._render_intensity(rb_n0)
    I1 = self._render_intensity(rb_n1)
    L0 = _linlog(I0, B)
    L1 = _linlog(I1, B)
    diff = torch.abs(L1 - L0) - C
    noev_loss = torch.clamp(diff, min=0.0).mean()
# CORRECT: Full pipeline from sampling to loss
```

Listing 264. C4 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Formula correct but no sampling
def no_event_loss(Lhat_k, Lhat_km1, C):
    return F.relu(torch.abs(Lhat_k - Lhat_km1) - C).mean()
# ERROR: Missing no-event location identification
```

Listing 265. C4 Implementation: GPT-5 (Score: 0.4)

```
# DeepSeek R1: Class-based implementation
class NoEventLoss:
    def __init__(self, C=0.2):
        self.C = C
    def __call__(self, pred_log_brightness1, pred_log_brightness2):
        brightness_diff = torch.abs(pred_log_brightness2 - pred_log_brightness1)
        return F.relu(brightness_diff - self.C).mean()
# CORRECT: Formula but missing sampling logic
```

Listing 266. C4 Implementation: DeepSeek R1 (Score: 0.8)

```
# Paper2Code: Not implemented
# Has no-event sampling but no loss computation
# CRITICAL ERROR: Missing no-event loss
```

Listing 267. C4 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: Not implemented
# CRITICAL ERROR: No loss implementation
```

Listing 268. C4 Implementation: AutoP2C (Score: 0.0)

Analysis: Only NERFIFY fully implements no-event loss with proper sampling. GPT-5 and DeepSeek R1 have correct formulas but miss the critical sampling component.

Component C5: Event + RGB Combined Loss - All Baselines

Combines supervision: $\mathcal{L} = \mathcal{L}_{\text{evs}} + \lambda_{\text{rgb}}\mathcal{L}_{\text{rgb}}$

```
# \nerfify\ : Integrated with Nerfacto pipeline
def get_loss_dict(self, outputs, batch, metrics_dict=None):
    loss_dict = super().get_loss_dict(outputs, batch, metrics_dict)
    # Scale RGB loss
    if "rgb_loss" in loss_dict and self.config.lambda_rgb != 1.0:
        loss_dict["rgb_loss"] = self.config.lambda_rgb * loss_dict["rgb_loss"]
    # Add event losses
    ev_loss, noev_loss = self._event_losses(batch)
    if ev_loss is not None:
        loss_dict["event_loss"] = ev_loss
# CORRECT: Proper loss combination with weights
```

Listing 269. C5 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Mathematical combination only
def combined_loss(Levs, Lrgb=None, lambda_rgb=1.0):
    if Lrgb is None:
        return Levs
    return Levs + lambda_rgb * Lrgb
# ERROR: No actual RGB rendering pipeline
```

Listing 270. C5 Implementation: GPT-5 (Score: 0.4)

```
# DeepSeek R1: Combined training step
def combined_training_step(self, event_batch, rgb_batch):
    event_loss = self.event_training_step(event_batch)
    rgb_loss = 0
    if rgb_batch is not None:
        # Simplified RGB loss computation
        rgb_loss = rgb_loss / len(rgb_batch)
    total_loss = event_loss + self.config.get('lambda_rgb', 1.0) * rgb_loss
# ISSUE: Incomplete RGB pipeline
```

Listing 271. C5 Implementation: DeepSeek R1 (Score: 0.6)

```
# Paper2Code: Not implemented
# CRITICAL ERROR: No combined loss logic
```

Listing 272. C5 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: Not implemented
# CRITICAL ERROR: No loss functions
```

Listing 273. C5 Implementation: AutoP2C (Score: 0.0)

Analysis: Only NERFIFY properly integrates event and RGB losses within a working framework. Others either lack RGB pipelines or miss the combination entirely.

Component C6: Event Pair Sampling Strategy - All Baselines

Samples event pairs within temporal windows (1ms average, 60ms window).

```python
# \nerfify\ : Integrated with DataManager
def _copy_event_tensors(self, src: Dict, dst: Dict) -> None:
    for k, v in src.items():
        if any(k.startswith(pref) for pref in EVENT_PREFIXES):
            dst[k] = v.to(self.device) if hasattr(v, "to") else v
# CORRECT: Full event batch handling
```

Listing 274. C6 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Synthetic dummy dataset only
class SyntheticEventDataset(Dataset):
    def __getitem__(self, idx):
        uk = torch.rand(2)
        tk, tk1 = torch.rand(1), torch.rand(1)
        pk = random.choice([-1, 1])
# ERROR: No real event sampling implementation
```

Listing 275. C6 Implementation: GPT-5 (Score: 0.2)

```python
# DeepSeek R1: Placeholder function
def sample_event_pairs(self, event_batch, max_time_window=60):
    # Implementation depends on your event data structure
    # This should sample neighboring events or events within time windows
    pass
# CRITICAL ERROR: Not implemented
```

Listing 276. C6 Implementation: DeepSeek R1 (Score: 0.3)

```python
# Paper2Code: Detailed sampling logic
def sample_event_pairs(self, n_samples: int) -> List[Dict[str, Any]]:
    for _ in range(n_samples):
        pixel = random.choice(valid_pixels)
        events_list = self.events_by_pixel[pixel]
        i = random.randint(0, len(events_list) - 2)
        # Accumulate polarities within window
        polarity_sum = sum([ev[1] for ev in events_list[i+1:chosen_index+1]])
# CORRECT: Good sampling but not integrated
```

Listing 277. C6 Implementation: Paper2Code (Score: 0.8)

```python
# AutoP2C: Not implemented
# CRITICAL ERROR: No event handling
```

Listing 278. C6 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY integrates event sampling with Nerfstudio's data pipeline. Paper2Code has detailed sampling logic but fails to integrate it. Others have minimal or no implementation.

Component C7: Camera Pose Interpolation - All Baselines

Uses spherical linear interpolation for rotation and cubic for translation.

```python
# \nerfify\ : Leverages Nerfstudio's built-in interpolation
# Handled by Nerfstudio's camera system
# CORRECT: Uses framework's optimized implementation
```

Listing 279. C7 Implementation: NERFIFY (Score: 0.9)

```python
# GPT-5: Correct slerp implementation
def slerp(q0, q1, t):
    dot = torch.sum(q0 * q1, dim=-1, keepdim=True)
    dot = torch.clamp(dot, -1.0, 1.0)
    theta_0 = torch.acos(dot)
    sin_theta_0 = torch.sin(theta_0)
    s0 = torch.sin(theta_0 - theta_t) / sin_theta_0
    s1 = torch.sin(theta_t) / sin_theta_0
    return s0 * q0 + s1 * q1
# ISSUE: Missing cubic interpolation for translation
```

Listing 280. C7 Implementation: GPT-5 (Score: 0.6)

```python
# DeepSeek R1: Not implemented
# Uses placeholder compute_rays function
# CRITICAL ERROR: No pose interpolation
```

Listing 281. C7 Implementation: DeepSeek R1 (Score: 0.0)

```python
# Paper2Code: Full implementation
def _interpolate_pose(self, timestamp: float) -> torch.Tensor:
    # Slerp for rotation
    q_interp = Utils.slerp(q0, q1, weight)
    R_interp = self.quaternion_to_matrix(q_interp)
    # Cubic for translation
    t_interp = Utils.cubic_interpolation(t0_vec, t1_vec, weight)
# CORRECT: Both slerp and cubic interpolation
```

Listing 282. C7 Implementation: Paper2Code (Score: 1.0)

```python
# AutoP2C: Not implemented
# CRITICAL ERROR: No pose handling
```

Listing 283. C7 Implementation: AutoP2C (Score: 0.0)

Analysis: Paper2Code surprisingly has the most complete pose interpolation. NERFIFY relies on Nerfstudio's implementation. GPT-5 has partial slerp. Others lack this component.

Component C8: Hash-based Encoding - All Baselines

Instant-NGP style multiresolution hash encoding with 16 levels.

```python
# \nerfify\ : Uses Nerfstudio's optimized hash encoding
# Configured in NerfactoField with proper parameters
# L=16, log2_hashmap=19, N_max=6144
# CORRECT: Framework's efficient implementation
```

Listing 284. C8 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Custom hash encoder implementation
class HashEncoder(nn.Module):
    def __init__(self, n_levels=16, n_features_per_level=2,
                 log2_hashmap_size=19):
        self.embeddings = nn.ParameterList([
            nn.Parameter(torch.randn(self.hash_size, n_features_per_level) * 1e-4)
            for _ in range(n_levels)
        ])
# CORRECT: Full implementation from scratch
```

Listing 285. C8 Implementation: GPT-5 (Score: 1.0)

```python
# DeepSeek R1: Uses tinycudann
self.encoding = tcnn.Encoding(
    n_input_dims=3,
    encoding_config={
        "otype": "HashGrid",
        "n_levels": 16,
        "n_features_per_level": 2,
        "log2_hashmap_size": 19
    }
)
# CORRECT: Efficient tinycudann implementation
```

Listing 286. C8 Implementation: DeepSeek R1 (Score: 1.0)

```python
# Paper2Code: Detailed hash grid implementation
class HashGridEncoder(nn.Module):
    def __init__(self, num_levels=16, dims_per_level=2,
                 hashmap_size_log2=19):
        for i in range(num_levels):
            resolution = N_min * ((N_max/N_min)**(i/(num_levels-1)))
            self.resolutions.append(resolution)
# CORRECT: Complete custom implementation
```

Listing 287. C8 Implementation: Paper2Code (Score: 1.0)

```python
# AutoP2C: Not implemented
# CRITICAL ERROR: No model at all
```

Listing 288. C8 Implementation: AutoP2C (Score: 0.0)

Analysis: All baselines except AutoP2C correctly implement hash encoding, showing this is a well-understood component. The difference lies in integration quality.

3.6.6. Scoring Analysis

Table 53. E-NeRF Implementation Scores

Method	C1	C2	C3	C4	C5	C6	C7	C8	Weighted Avg$_{\text{LLM}}$
NERFIFY	1.0	1.0	1.0	1.0	1.0	1.0	0.9	1.0	1.00
GPT-5	0.8	1.0	0.8	0.4	0.4	0.2	0.6	1.0	0.60
DeepSeek R1	0.9	1.0	1.0	0.8	0.6	0.3	0.0	1.0	0.72
Paper2Code	0.5	1.0	0.0	0.0	0.0	0.8	1.0	1.0	0.48
AutoP2C	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.05

3.6.7. Why Baselines Fail Despite Component Scores

GPT-5 (Score: 0.60 components, 0% trainable)

- **Strengths**: Near-perfect mathematical implementations, clean modular code
- **Fatal Issues**:
  - No Nerfstudio integration - standalone PyTorch implementation
  - Missing event data pipeline - uses synthetic dummy data
  - Training loop runs only 3 demonstration epochs then terminates
- **Result**: Code compiles but cannot train on real event data

DeepSeek R1 (Score: 0.72 components, 0% trainable)

- **Strengths**: Comprehensive component coverage, uses efficient tinycudann
- **Fatal Issues**:
  - Event sampling returns empty - placeholder function not implemented
  - Missing camera pose interpolation entirely
  - No actual training step - only function signatures
- **Result**: Most complete baseline but critical functions are stubs

Paper2Code (Score: 0.48 components, 0% trainable)

- **Strengths**: Extensive 1650-line codebase, detailed event sampling logic
- **Fatal Issues**:
  - Import errors - references non-existent colmap Python module
  - Missing normalized event loss variant
  - Event sampling not connected to training loop
- **Result**: Extensive boilerplate that doesn't execute

AutoP2C (Score: 0.05 components, 0% trainable)

- **Strengths**: Basic dataset structure attempted
- **Fatal Issues**:
  - No model implementation whatsoever
  - No loss functions defined
  - Only contains data loading placeholders
- **Result**: Complete failure - non-functional code

3.6.8. Hyperparameter Fidelity

Table 54. Hyperparameter Implementation Accuracy

Parameter	Paper	NERFIFY	GPT-5	R1	P2C	AutoP2C
Contrast threshold C	0.2	✓	✓	✓	✓	×
Linlog threshold B	20.0	✓	✓	✓	✓	×
Batch size	30096	4096*	8	Variable	30096	×
Hash levels L	16	✓	✓	✓	✓	×
$\log_2$ hashmap	19	✓	✓	✓	✓	×
$N_{\max}$	$B \cdot 2048$	✓	✓	Variable	✓	×
Learning rate	1e-3	1e-2*	✓	✓	✓	×
W Score	–	0.95	0.75	0.80	0.60	0.00

*NERFIFY adapts parameters for Nerfstudio optimization

3.6.9. Conclusion

All baselines attempt to implement E-NeRF with varying sophistication, from AutoP2C's complete failure (0.05 score) to DeepSeek R1's comprehensive but non-functional attempt (0.72 score). GPT-5 achieves mathematically correct components but lacks system integration, while Paper2Code generates extensive boilerplate missing critical training logic. Only NERFIFY produces immediately trainable code by properly integrating event-specific losses into Nerfstudio's proven NeRF framework, achieving perfect coverage (1.00 score) and full functionality. This stark contrast—0% trainable rate for all baselines versus 100% for NERFIFY —demonstrates that effective paper-to-code translation for event-based vision demands deep domain specialization over generic approaches.

3.7. StyleNeRF: A Style-Based 3D-Aware Generator for High-Resolution Image Synthesis

3.7.1. Paper Overview

StyleNeRF [7] introduces a groundbreaking 3D-aware generative model that integrates neural radiance fields with style-based generation for high-resolution image synthesis. The paper addresses the computational bottleneck in NeRF-based image generation by proposing an efficient approximation that aggregates features in 2D before final color computation, enabling progressive upsampling. This approach achieves both high visual quality and 3D consistency while reducing rendering time from minutes to seconds.

3.7.2. Implementation Overview

Table 55. StyleNeRF Implementation Summary Across All Baselines

Aspect	NERFIFY	GPT-5	DeepSeek R1	Paper2Code	AutoP2C
Lines of Code	742	385	412	1847	623
File Organization	Plugin	Single	Single	Multi-file	Multi-file
Style Modulation	✓	Simplified	✓	×	×
2D Upsampling Path	✓	Partial	✓	×	×
NeRF-path Regularization	✓	Wrong	×	×	×
Custom Upsampler	✓	∼	Partial	×	×
View Direction Removal	✓	×	×	×	×
Nerfstudio Integration	✓	×	×	×	×
Trainable	✓	×	×	×	×

Note: NERFIFY always produces trainable code as a complete Nerfstudio plugin

3.7.3. Novel Components

Table 56. Novel Components in StyleNeRF with Importance Weights

ID	Component	Weight w_i
C1	Style-conditioned NeRF field (Eq. 2-3)	0.20
C2	Fourier positional encoding (Eq. 1)	0.08
C3	Low-res feature rendering + 2D upsampling (Eq. 5-6)	0.22
C4	Custom upsampler with blur kernel (Eq. 7)	0.15
C5	NeRF-path regularization loss (Eq. 9)	0.15
C6	View direction removal for consistency	0.05
C7	Mapping network (StyleGAN2-based)	0.05
C8	NeRF++ foreground/background split	0.05
C9	Progressive training strategy (3-stage)	0.03
C10	Hyperparameter fidelity	0.02

3.7.4. Quantitative Metrics

3.7.5. Component-by-Component Analysis

Component C1: Style-Conditioned NeRF Field - All Baselines

StyleNeRF introduces style modulation for NeRF MLPs using FiLM-style conditioning:

$$\phi_w^n(\mathbf{x}) = g_w^n \circ g_w^{n-1} \circ \ldots \circ g_w^1 \circ \zeta(\mathbf{x})$$

where g_w^i are style-modulated layers with weight matrices modulated by style vector $\mathbf{w}$.

Table 57. StyleNeRF Implementation Coverage Metrics

Method	C	I	M	W	$\text{Score}_{\text{LLM}}$
NERFIFY (Ours)	1.00	0.00	0.00	1.00	0.98
GPT-5	0.40	0.30	0.30	0.55	0.52
DeepSeek R1	0.50	0.30	0.20	0.64	0.62
Paper2Code	0.30	0.40	0.30	0.46	0.28
AutoP2C	0.00	0.10	0.90	0.00	0.00

```python
# \nerfify\ : Complete FiLM-style modulation with per-layer affine transforms
class StyleMLP(nn.Module):
    def __init__(self, in_dim, width, depth, style_dim):
        super().__init__()
        self.layers = nn.ModuleList([nn.Linear(...) for _ in range(depth)])
        self.style_affines = nn.ModuleList([
            nn.Linear(style_dim, 2 * width) for _ in range(depth)
        ])
    def forward(self, x, style):
        for lin, aff in zip(self.layers, self.style_affines):
            gb = aff(style)
            gamma, beta = torch.chunk(gb, 2, dim=-1)
            x = (1.0 + gamma) * lin(x) + beta  # FiLM modulation
            x = self.act(x)
        return x
```

Listing 289. C1 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Simplified modulation missing +1 formulation
class ModulatedMLP(nn.Module):
    def forward(self, x, w):
        s = self.style(w)
        x = x * (s.unsqueeze(1) + 1e-6)  # ERROR: Should be (1 + s)
        x = self.linear(x)
        if self.demodulate:
            d = torch.rsqrt((x ** 2).mean(dim=-1, keepdim=True) + 1e-8)
            x = x * d
        return self.act(x)
```

Listing 290. C1 Implementation: GPT-5 (Score: 0.6)

```python
# DeepSeek R1: Correct modulation with demodulation
class StyleLinear(nn.Module):
    def forward(self, x, style):
        style = self.affine(style)
        weight = self.weight.unsqueeze(0)
        weight = weight * (style.unsqueeze(1) + 1)  # CORRECT: (s + 1)
        if self.demodulate:
            d = torch.rsqrt((weight ** 2).sum(dim=2, keepdim=True) + 1e-8)
            weight = weight * d
        return (x.unsqueeze(1) * weight).sum(dim=2) + self.bias
```

Listing 291. C1 Implementation: DeepSeek R1 (Score: 1.0)

```python
# Paper2Code: No style modulation at all
class StyleNerfField(nn.Module):
    def __init__(self, hidden_units=256):
        super().__init__()
        # CRITICAL ERROR: Standard MLP without style conditioning
```

```
        self.density_network = nn.Sequential(
            nn.Linear(63, hidden_units),
            nn.ReLU(),
            nn.Linear(hidden_units, 1)
        )
        # MISSING: Style vector integration entirely
```

Listing 292. C1 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: No model implementation at all
# CRITICAL ERROR: Only dataset loading code present
class AFHQDataset(data.Dataset):
    def __init__(self, data_path, image_size=512):
        # MISSING: Entire model, no style modulation implemented
        self.dataset = []
```

Listing 293. C1 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY and DeepSeek R1 correctly implement FiLM-style modulation with proper $(1 + \gamma)$ formulation. GPT-5 has the structure but uses incorrect modulation. Paper2Code and AutoP2C completely miss this core innovation.

Component C2: Fourier Positional Encoding - All Baselines

The paper uses standard NeRF positional encoding with $L = 10$ for positions and $L = 4$ for directions:

$$\zeta^L(x) = [\sin(2^0 x), \cos(2^0 x), ..., \sin(2^{L-1} x), \cos(2^{L-1} x)]$$

```
# \nerfify\ : Correct Fourier encoding with proper frequencies
self.pos_encoding = NeRFEncoding(
    in_dim=3, num_frequencies=10,  # L=10 for positions
    min_freq_exp=0.0, max_freq_exp=9.0, include_input=True
)
self.dir_encoding = NeRFEncoding(
    in_dim=3, num_frequencies=4,   # L=4 for directions
    min_freq_exp=0.0, max_freq_exp=3.0, include_input=True
)
```

Listing 294. C2 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Correct implementation
class FourierEmbedding(nn.Module):
    def __init__(self, num_freqs=10):
        super().__init__()
        self.freq_bands = 2.0 ** torch.linspace(0, num_freqs - 1, num_freqs)
    def forward(self, x):
        out = [x]
        for f in self.freq_bands:
            out.append(torch.sin(f * x))
            out.append(torch.cos(f * x))
        return torch.cat(out, dim=-1)  # CORRECT
```

Listing 295. C2 Implementation: GPT-5 (Score: 1.0)

```
# DeepSeek R1: Correct encoding
class FourierFeatureEncoding(nn.Module):
    def __init__(self, L=10, include_input=True):
        super().__init__()
        self.L = L
        self.include_input = include_input
        self.output_dim = 2 * L * 3 * 2 + (3 if include_input else 0)
```

Listing 296. C2 Implementation: DeepSeek R1 (Score: 1.0)

```
# Paper2Code: Correct formula, wrong usage
def positional_encoding(x: torch.Tensor, L: int) -> torch.Tensor:
    encoding = [x]
    for i in range(L):
        for fn in [torch.sin, torch.cos]:
            encoding.append(fn((2.0 ** i) * x))
    return torch.cat(encoding, dim=-1)
    # ISSUE: Not integrated into model properly
```

Listing 297. C2 Implementation: Paper2Code (Score: 0.8)

```
# AutoP2C: No positional encoding
# CRITICAL ERROR: Missing entirely
```

Listing 298. C2 Implementation: AutoP2C (Score: 0.0)

Analysis: Most baselines correctly implement Fourier encoding as it's a standard NeRF component. Only AutoP2C completely misses it.

Component C3: Low-Resolution Feature Rendering + 2D Upsampling - All Baselines

The core efficiency innovation renders features at 32×32 resolution then upsamples:

$$I_w^{\text{Approx}}(\mathbf{r}) \approx h_c \circ [\phi_w^{n_c, n_\sigma}(\mathcal{A}(\mathbf{r})), \zeta(\mathbf{d})]$$

```
# \nerfify\ : Complete 2D aggregation for efficient upsampling
def get_outputs(self, ray_bundle):
    # Low-res volumetric rendering
    field_outputs = self.field(ray_samples)
    weights = ray_samples.get_weights(sigmas)
    # Aggregate features in 2D (early aggregation)
    style_feat = field_outputs["style_feat"]  # [N, S, C]
    feat_agg = torch.sum(weights * style_feat, dim=-2)  # [N, C]
    # Decode once from aggregated features
    approx_rgb = self.field.decode_color_from_features(
        features=feat_agg, directions=ray_bundle.directions
    )
    # Then upsample via synthesis network
```

Listing 299. C3 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Partial - has low-res rendering but wrong upsampling
def forward(self, z, camera_poses, render_full=True):
    rays_o, rays_d = self.generate_rays(camera_poses, res=32)
    feature_map, _, _ = self.generator.volume_render(rays_o, rays_d, styles)
    # ISSUE: Missing proper 2D aggregation pipeline
    for block in self.synthesis_blocks:
        x = block(x, w)  # Direct upsampling without proper aggregation
```

Listing 300. C3 Implementation: GPT-5 (Score: 0.6)

```python
# DeepSeek R1: Correct approach with progressive upsampling
def forward(self, z, camera_poses, render_full=True):
    if not render_full:
        rays_o, rays_d = self.generate_rays(camera_poses, res=32)
        feature_map, depth_map, weights = self.generator.volume_render(
            rays_o, rays_d, styles)
        return feature_map, depth_map
    else:
        # Progressive upsampling from low-res features
        feature_map = feature_map.reshape(B, 32, 32, C)
        for upsample_layer, block in zip(self.upsample_layers, self.blocks):
            feature_map = upsample_layer(feature_map, styles)
```

Listing 301. C3 Implementation: DeepSeek R1 (Score: 0.8)

```python
# Paper2Code: Direct high-res rendering (wrong approach)
def forward(self, rays_o, rays_d, style):
    # CRITICAL ERROR: Renders directly at target resolution
    points = sample_points_on_rays(rays_o, rays_d, n_samples=64)
    densities = self.density_network(points)  # Expensive!
    colors = self.color_network(points)
    rgb = volume_render(densities, colors, points)
    # MISSING: Low-res rendering + upsampling strategy entirely
```

Listing 302. C3 Implementation: Paper2Code (Score: 0.0)

```python
# AutoP2C: No rendering implementation
# CRITICAL ERROR: Missing entirely
```

Listing 303. C3 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY correctly implements 2D feature aggregation for efficient upsampling. DeepSeek R1 has the right approach. GPT-5 partially implements it. Paper2Code and AutoP2C miss this critical efficiency innovation.

Component C4: Custom Upsampler with Blur Kernel - All Baselines

StyleNeRF's hybrid upsampler combines pixel-shuffle with learnable MLPs and fixed blur:

$$\text{Upsample}(X) = \text{Conv2d}(\text{PixelShuffle}(\text{Repeat}(X, 4) + \psi_\theta(X), 2), K)$$

```python
# \nerfify\ : Complete hybrid upsampler
class HybridUpsampler(nn.Module):
    def forward(self, x):
        x_repeat = x.repeat_interleave(4, dim=1)  # Repeat 4x
        psi = self.mlp(x.permute(0,2,3,1))  # Learnable variations
        x_combined = x_repeat + psi.permute(0,3,1,2)
        x_shuffled = F.pixel_shuffle(x_combined, 2)  # PixelShuffle
        x_blurred = self.blur_conv(x_shuffled)  # Fixed blur kernel
        return x_blurred
```

Listing 304. C4 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Partial - missing blur kernel
class Upsampler(nn.Module):
    def forward(self, x):
        x_rep = x.repeat(1, 4, 1, 1)
        psi = self.mlp(x.permute(0, 2, 3, 1)).permute(0, 3, 1, 2)
        x = F.pixel_shuffle(x_rep + psi, 2)
        # MISSING: Fixed blur convolution
        return x  # ERROR: No blur kernel applied
```

Listing 305. C4 Implementation: GPT-5 (Score: 0.4)

```python
# DeepSeek R1: Has blur but simplified
class CustomUpsampler(nn.Module):
    def __init__(self, in_ch, out_ch, style_dim):
        super().__init__()
        self.mlp = nn.Sequential(StyleLinear(...))
        self.blur = nn.Conv2d(in_ch, out_ch, 3, padding=1, bias=False)
        nn.init.dirac_(self.blur.weight)  # Initialize as identity
    def forward(self, x, style):
        x_repeated = F.pixel_shuffle(x.repeat(1, 4, 1, 1), 2)
        x_mlp = self.mlp(x_permuted, style)
        x_combined = x_repeated + F.pixel_shuffle(x_mlp, 2)
        return self.blur(x_combined)  # Apply blur
```

Listing 306. C4 Implementation: DeepSeek R1 (Score: 0.8)

```python
# Paper2Code: No custom upsampler
# CRITICAL ERROR: Missing entirely, using standard upsampling
```

Listing 307. C4 Implementation: Paper2Code (Score: 0.0)

```python
# AutoP2C: No upsampler implementation
# CRITICAL ERROR: Missing entirely
```

Listing 308. C4 Implementation: AutoP2C (Score: 0.0)

Analysis: Only NERFIFY fully implements the hybrid upsampler with all components. DeepSeek R1 has most elements. GPT-5 misses the blur kernel. Paper2Code and AutoP2C don't implement it.

Component C5: NeRF-Path Regularization Loss - All Baselines

Novel loss enforcing 3D consistency:

$$\mathcal{L}_{\text{NeRF-path}} = \frac{1}{|S|} \sum_{(i,j)\in S} (I_w^{\text{Approx}}[i,j] - I_w^{\text{NeRF}}[i,j])^2$$

```python
# \nerfify\ : Correct NeRF-path regularization
def get_loss_dict(self, outputs, batch):
    if self.config.nerf_path_loss_mult > 0.0:
        rgb_full = outputs["rgb"]  # Full volumetric RGB
        rgb_approx = outputs["approx_rgb"]  # Aggregated-feature RGB
        # Subsample pixels (25% by default)
        num = int(self.config.nerf_path_sample_frac * rgb_full.shape[0])
        idx = torch.randperm(rgb_full.shape[0])[:num]
        nerf_path = F.mse_loss(rgb_approx[idx], rgb_full[idx])
        loss_dict["nerf_path_reg"] = 0.2 * nerf_path  #    = 0.2
```

Listing 309. C5 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Wrong formulation entirely
def nerf_path_reg(fake_high, fake_low):
    # CRITICAL ERROR: Comparing high-res vs downsampled
    # Should be full volumetric vs aggregated features!
    return F.mse_loss(fake_high, fake_low.detach())
# In training:
reg = 0.2 * nerf_path_reg(fake_img,
    F.interpolate(fake_img, scale_factor=0.5))  # WRONG!
```

Listing 310. C5 Implementation: GPT-5 (Score: 0.2)

```python
# DeepSeek R1: Missing NeRF-path regularization
# CRITICAL ERROR: Not implemented
class NeRFPathRegularization(nn.Module):
    def forward(self, approx_output, nerf_output, mask=None):
        # ERROR: Defined but never used in training loop
```

Listing 311. C5 Implementation: DeepSeek R1 (Score: 0.0)

```python
# Paper2Code: No NeRF-path regularization
def nerf_path_regularization(predictions, targets):
    # CRITICAL ERROR: Wrong concept - not the paper's formulation
    squared_differences = (sampled_predictions - sampled_targets) ** 2
    # MISSING: Comparison between full and approx paths
```

Listing 312. C5 Implementation: Paper2Code (Score: 0.0)

```python
# AutoP2C: No loss implementation
# CRITICAL ERROR: Missing entirely
```

Listing 313. C5 Implementation: AutoP2C (Score: 0.0)

Analysis: Only NERFIFY correctly implements the NeRF-path regularization comparing full volumetric vs aggregated-feature RGB. GPT-5 implements a completely wrong loss. Others don't implement it at all.

Component C6: View Direction Removal - All Baselines

Paper removes view direction conditioning from color prediction to improve 3D consistency.

```python
# \nerfify\ : Correctly removes view direction
def __init__(self, remove_viewdir: bool = True):
    self.remove_viewdir = remove_viewdir
def _encode_directions(self, directions):
    if self.remove_viewdir:
        # Return empty tensor - no view direction
        return torch.zeros((*directions.shape[:-1], 0),
                           device=directions.device)
    return self.dir_encoding(directions)
```

Listing 314. C6 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Still uses view direction
self.color_out = nn.Linear(hidden_dim, 3)
# ERROR: Color prediction still depends on view direction
# MISSING: remove_viewdir flag and implementation
```

Listing 315. C6 Implementation: GPT-5 (Score: 0.0)

```python
# DeepSeek R1: No view direction removal
def forward_color(self, x, style, is_background=False):
    # ERROR: Still conditions on view direction implicitly
    color = self.color_head(h)  # No removal implemented
```

Listing 316. C6 Implementation: DeepSeek R1 (Score: 0.0)

```python
# Paper2Code: No view direction removal
# MISSING: Not implemented
```

Listing 317. C6 Implementation: Paper2Code (Score: 0.0)

```python
# AutoP2C: No model implementation
# MISSING: Entirely
```

Listing 318. C6 Implementation: AutoP2C (Score: 0.0)

Analysis: Only NERFIFY implements view direction removal. All baselines miss this 3D consistency improvement.

Component C7: Mapping Network - All Baselines

StyleGAN2-style mapping from z to w with 8 FC layers.

```python
# \nerfify\ : Complete mapping network
self.mapping_network = MappingNetwork(
    latent_dim=512, style_dim=512,
    num_layers=8, lr_multiplier=0.01  # 100x lower LR
)
```

Listing 319. C7 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Correct mapping network
class MappingNetwork(nn.Module):
    def __init__(self, z_dim=512, w_dim=512, n_layers=8):
        layers = []
        for i in range(n_layers):
            layers.append(nn.Linear(z_dim if i==0 else w_dim, w_dim))
            layers.append(nn.LeakyReLU(0.2))
        self.mapping = nn.Sequential(*layers)
```

Listing 320. C7 Implementation: GPT-5 (Score: 1.0)

```python
# DeepSeek R1: Correct implementation
class MappingNetwork(nn.Module):
    def forward(self, z):
        z = F.normalize(z, dim=-1)  # Normalize input
        for layer in self.layers:
            z = F.leaky_relu(layer(z), 0.2)
        return z
```

Listing 321. C7 Implementation: DeepSeek R1 (Score: 1.0)

```python
# Paper2Code: Has mapping but wrong integration
class MappingNetwork(nn.Module):
    def __init__(self, latent_dim, style_dim, num_layers):
        # ISSUE: Correct structure but not properly used
        self.mapping = nn.Sequential(*layers)
```

Listing 322. C7 Implementation: Paper2Code (Score: 0.6)

```python
# AutoP2C: No mapping network
# MISSING: Entirely
```

Listing 323. C7 Implementation: AutoP2C (Score: 0.0)

Analysis: Most baselines implement the mapping network correctly as it's a standard StyleGAN2 component.

Component C8: NeRF++ Foreground/Background Split - All Baselines

NeRF++ architecture with inverted sphere parameterization for background.

```
# \nerfify\ : Complete NeRF++ implementation
# Foreground in unit sphere, background with inverted sphere
self.spatial_distortion = None  # Use scene box normalization
# Handles fg/bg split properly in field implementation
```

Listing 324. C8 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Correct fg/bg split
# Foreground network
self.field = StyleNeRFField(hidden_dim=256)
# Background network (smaller)
self.bg_field = StyleNeRFField(hidden_dim=128)
```

Listing 325. C8 Implementation: GPT-5 (Score: 1.0)

```
# DeepSeek R1: Correct implementation
def forward_geometry(self, x, style, is_background=False):
    if is_background:
        r = torch.sqrt((x ** 2).sum(dim=-1, keepdim=True))
        x_inv = torch.cat([x / r, 1.0 / r], dim=-1)  # Inverted sphere
```

Listing 326. C8 Implementation: DeepSeek R1 (Score: 1.0)

```
# Paper2Code: Mentions but doesn't implement properly
# ISSUE: No actual fg/bg separation in rendering
```

Listing 327. C8 Implementation: Paper2Code (Score: 0.4)

```
# AutoP2C: No NeRF++ implementation
# MISSING: Entirely
```

Listing 328. C8 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY , GPT-5, and DeepSeek R1 correctly implement NeRF++. Paper2Code mentions it but doesn't implement properly.

Component C9: Progressive Training Strategy - All Baselines

Three-stage training: 500k low-res, 5M progressive, 25M high-res.

```
# \nerfify\ : Has progressive structure
self.stage1_images = 500000
self.stage2_images = 5000000
self.stage3_images = 25000000
# Implemented through config but simplified
```

Listing 329. C9 Implementation: NERFIFY (Score: 0.8)

```
# GPT-5: No progressive training
# ERROR: Trains at target resolution from start
for iteration in range(num_iterations):
    # MISSING: Resolution scheduling
```

Listing 330. C9 Implementation: GPT-5 (Score: 0.0)

```
# DeepSeek R1: Has stages but incomplete
stages = [(32, 500000), (1024, 5000000), (1024, 25000000)]
# ISSUE: Structure exists but not fully implemented
```

Listing 331. C9 Implementation: DeepSeek R1 (Score: 0.4)

```
# Paper2Code: Defines stages but wrong usage
class ProgressiveTrainingConfig:
    stage1_images: int = 500_000
    # ISSUE: Not integrated with resolution changes
```

Listing 332. C9 Implementation: Paper2Code (Score: 0.6)

```
# AutoP2C: No training implementation
# MISSING: Entirely
```

Listing 333. C9 Implementation: AutoP2C (Score: 0.0)

Analysis: No baseline fully implements progressive training. NERFIFY has the structure. Paper2Code defines it but doesn't use it properly.

Component C10: Hyperparameter Fidelity - All Baselines

Matching paper's exact hyperparameters for reproducibility.

```
# \nerfify\ : All hyperparameters match paper
style_dim=512, trunk_layers=4, trunk_hidden_dim=256,
color_layers=2, color_hidden_dim=128,
nerf_path_loss_mult=0.2, nerf_path_sample_frac=0.25
```

Listing 334. C10 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Some parameters correct
style_dim=512, hidden_dim=256
# ISSUE: Missing several key parameters
```

Listing 335. C10 Implementation: GPT-5 (Score: 0.6)

```
# DeepSeek R1: Most parameters correct
style_dim=512, hidden_dim=256, bg_hidden_dim=128
# ISSUE: Missing nerf_path parameters
```

Listing 336. C10 Implementation: DeepSeek R1 (Score: 0.7)

```
# Paper2Code: Partial match
latent_dim: int = 512
# ISSUE: Many incorrect or missing
```

Listing 337. C10 Implementation: Paper2Code (Score: 0.5)

```
# AutoP2C: No hyperparameters
# MISSING: Entirely
```

Listing 338. C10 Implementation: AutoP2C (Score: 0.0)

Analysis: Only NERFIFY matches all hyperparameters exactly. Others have partial matches.

Table 58. StyleNeRF Implementation Scores

Method	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	Weighted Avg$_{LLM}$
NERFIFY	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.8	1.0	0.98
GPT-5	0.6	1.0	0.6	0.4	0.2	0.0	1.0	1.0	0.0	0.6	0.52
DeepSeek R1	1.0	1.0	0.8	0.8	0.0	0.0	1.0	1.0	0.4	0.7	0.62
Paper2Code	0.0	0.8	0.0	0.0	0.0	0.0	0.6	0.4	0.6	0.5	0.28
AutoP2C	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00

3.7.6. Scoring Analysis

3.7.7. Why Baselines Fail Despite Component Scores

GPT-5 (Score: 0.52 components, 0% trainable)

- **Strengths**: Correct Fourier encoding, mapping network, NeRF++ split
- **Fatal Issues**:
  - Wrong NeRF-path regularization (compares high-res vs downsampled instead of volumetric vs aggregated)
  - No Nerfstudio integration - standalone implementation incompatible with benchmark
  - Missing view direction removal and blur kernel in upsampler
- **Result**: Cannot train on benchmark datasets, incorrect 3D consistency enforcement

DeepSeek R1 (Score: 0.62 components, 0% trainable)

- **Strengths**: Perfect style modulation with demodulation, correct NeRF++ architecture
- **Fatal Issues**:
  - Missing NeRF-path regularization entirely - loses key 3D consistency benefit
  - No Nerfstudio plugin structure - cannot integrate with training pipeline
  - Missing view direction removal
- **Result**: Strong mathematical understanding but incomplete system, cannot execute training

Paper2Code (Score: 0.28 components, 0% trainable)

- **Strengths**: Configuration management, dataset loading structure
- **Fatal Issues**:
  - No style modulation - uses standard MLPs without conditioning
  - Missing 2D upsampling path - renders directly at high resolution (computationally infeasible)
  - No NeRF-path regularization or custom upsampler
- **Result**: Fundamental misunderstanding of StyleNeRF architecture, cannot scale beyond 256×256

AutoP2C (Score: 0.00 components, 0% trainable)

- **Strengths**: Attempted dataset loading (with incorrect URLs)
- **Fatal Issues**:
  - No model implementation whatsoever
  - Only generates dataset and evaluation code
  - Import errors prevent execution
- **Result**: Complete failure to implement any StyleNeRF components

3.7.8. Hyperparameter Fidelity

Table 59. Hyperparameter Implementation Accuracy

Parameter	Paper	NERFIFY	GPT-5	R1	P2C	AutoP2C
Style dim	512	✓	✓	✓	✓	×
Latent dim	512	✓	✓	✓	✓	×
Mapping layers	8	✓	✓	✓	✓	×
FG hidden dim	256	✓	✓	✓	✓	×
BG hidden dim	128	✓	✓	✓	256	×
L_{pos}	10	✓	✓	✓	✓	×
L_{dir}	4	✓	✓	✓	✓	×
Initial resolution	32	✓	✓	✓	64	×
β (NeRF-path)	0.2	✓	Wrong	×	×	×
λ (R1 reg)	0.5	✓	×	×	0.5	×
W Score	–	1.00	0.55	0.64	0.46	0.00

3.7.9. Conclusion

All baselines attempt to implement StyleNeRF with varying degrees of sophistication. GPT-5 and DeepSeek R1 demonstrate understanding of core architectural concepts but fail on critical details - GPT-5 implements the wrong NeRF-path regularization while DeepSeek R1 omits it entirely. Paper2Code fundamentally misunderstands the method, implementing standard NeRF without style conditioning or the efficiency innovations. AutoP2C completely fails, producing only non-functional dataset code. Despite some baselines achieving reasonable component scores (DeepSeek R1: 0.62, GPT-5: 0.52), none produce trainable code due to missing Nerfstudio integration, incorrect loss formulations, or architectural failures. Only NERFIFY produces immediately trainable code as a complete Nerfstudio plugin, demonstrating the critical importance of domain-specific synthesis with a stark 100% vs 0% trainable rate highlighting the necessity of specialized paper-to-code approaches for complex vision research.

3.8. iNeRF: Inverting Neural Radiance Fields for Pose Estimation

3.8.1. Paper Overview

iNeRF [31] introduces a framework for 6DoF pose estimation by inverting a trained NeRF model through gradient-based optimization. The method performs mesh-free pose refinement by backpropagating photometric loss through differentiable volume rendering, optimizing camera poses on the SE(3) manifold. Key innovations include interest-region ray sampling for computational efficiency, exponential map parameterization for valid pose updates, and self-supervised NeRF training by estimating poses for unlabeled images.

Table 60. iNeRF Implementation Summary Across All Baselines

Aspect	NERFIFY	GPT-5	DeepSeek R1	Paper2Code	AutoP2C
Lines of Code	580	180	650	2100	950
File Organization	Plugin	Single	Single	Multi-file	Multi-file
SE(3) Exp Map	✓	Partial	✓	✓	×
Interest Sampling	✓	Simplified	✓	✓	×
YUV Loss	✓	×	✓	✓	×
LR Schedule	✓	×	Wrong	✓	×
Nerfstudio Integration	✓	×	×	×	×
Trainable	✓	×	×	×	×

Note: NERFIFY always produces trainable code as a complete Nerfstudio plugin

Table 61. Novel Components in iNeRF with Importance Weights

ID	Component	Weight w_i
C1	SE(3) exponential map parameterization	0.20
C2	Interest region sampling with dilation	0.18
C3	Interest point sampling (corner detection)	0.12
C4	Photometric RGB MSE loss	0.10
C5	YUV-UV loss variant (LineMOD)	0.08
C6	Camera LR schedule ($\alpha_t = \alpha_0 \cdot 0.8^{t/100}$)	0.10
C7	Small ray batch (2048 rays)	0.08
C8	Gradient-based pose updates	0.07
C9	Volume rendering integration	0.04
C10	Pose-only optimization mode	0.03

Table 62. iNeRF Implementation Coverage Metrics

Method	C	I	M	W	$\text{Score}_{\text{LLM}}$
NERFIFY (Ours)	1.00	0.00	0.00	1.00	0.97
GPT-5	0.50	0.30	0.20	0.60	0.58
DeepSeek R1	0.60	0.30	0.10	0.70	0.68
Paper2Code	0.70	0.20	0.10	0.80	0.75
AutoP2C	0.00	0.10	0.90	0.00	0.05

3.8.2. Implementation Overview

3.8.3. Novel Components

3.8.4. Quantitative Metrics

3.8.5. Component-by-Component Analysis

Component C1: SE(3) Exponential Map Parameterization - All Baselines

The paper specifies pose updates via the exponential map from Lie algebra $\mathfrak{se}(3)$ to SE(3): $T = \exp([\xi]_\times)T_0$ where $\xi = [\omega, v]^T \in \mathbb{R}^6$ and the rotation matrix uses Rodrigues' formula: $R = I + \frac{\sin\theta}{\theta}[\omega]_\times + \frac{1-\cos\theta}{\theta^2}[\omega]_\times^2$.

```
# \nerfify\ : Uses Nerfstudio's camera optimizer with correct SE(3)
camera_optimizer=CameraOptimizerConfig(mode="SO3xR3")
# In callbacks:
def set_camera_lr(step: int) -> None:
    T_update = se3_exponential_map(self.pose_params)
    T_current = T_update @ self.initial_pose
# CORRECT: Full SE(3) manifold parameterization
```

Listing 339. C1 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: SE(3) exp map without small angle handling
def se3_expmap(screw):
    omega = screw[:, :3]; v = screw[:, 3:]
    theta = torch.norm(omega, dim=1, keepdim=True) + 1e-8
    # MISSING: No small angle approximation
    R = I + torch.sin(theta) * omega_hat / theta + (1 - torch.cos(theta)) * (omega_hat @ omega_hat) / theta**2
    K = I * theta + (1 - torch.cos(theta)) * omega_hat
    t = (K @ v[..., None]).squeeze(-1)
# ERROR: Numerical instability for small rotations
```

Listing 340. C1 Implementation: GPT-5 (Score: 0.8)

```
# DeepSeek R1: Complete SE(3) with small angle handling
def se3_exp_map(self, screw_axis):
    omega = screw_axis[:3]; v = screw_axis[3:]
    theta = torch.norm(omega)
    if theta < 1e-8:  # CORRECT: Small angle approximation
        R = torch.eye(3) + self.skew_symmetric(omega)
        t = v
    else:
        # CORRECT: Full Rodrigues formula
        omega_hat = omega / theta
        omega_hat_skew = self.skew_symmetric(omega_hat)
        R = torch.eye(3) + torch.sin(theta) * omega_hat_skew +
            (1 - torch.cos(theta)) * omega_hat_skew @ omega_hat_skew
        V = (torch.sin(theta)/theta) * torch.eye(3) + ...
        t = V @ v.unsqueeze(-1).squeeze(-1)
```

Listing 341. C1 Implementation: DeepSeek R1 (Score: 1.0)

```
# Paper2Code: Textbook-correct SE(3) implementation
def se3_exponential_map(xi: torch.Tensor) -> torch.Tensor:
    omega = xi[0:3]; nu = xi[3:6]
    theta = torch.norm(omega)
    eps: float = 1e-6
    if theta.item() < eps:  # CORRECT: Small angle handling
        R = I + A; t = nu
    else:
        sin_theta = torch.sin(theta); cos_theta = torch.cos(theta)
        R = I + (sin_theta/theta)*A + ((1-cos_theta)/theta**2)*A2
        t = (I*theta + (1-cos_theta)*A + (theta-sin_theta)*A2) @ (nu/theta)
    T = torch.eye(4); T[0:3, 0:3] = R; T[0:3, 3] = t
# CORRECT: Complete SE(3) with proper parameterization
```

Listing 342. C1 Implementation: Paper2Code (Score: 1.0)

```
# AutoP2C: No SE(3) parameterization at all
def optimize_pose(self, image, initial_pose):
    pose = initial_pose.clone().detach().requires_grad_(True)
    optimizer = optim.Adam([pose], lr=0.01)
    # CRITICAL ERROR: Direct matrix optimization
    for _ in range(100):
        loss.backward()
        optimizer.step()
        with torch.no_grad():
            pose[:] = self.project_to_se3(pose)  # Empty placeholder
def project_to_se3(self, pose):
```

```
    return pose  # WRONG: No actual projection
```

Listing 343. C1 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY , DeepSeek R1, and Paper2Code correctly implement the full SE(3) exponential map with proper Rodrigues formula and small angle handling. GPT-5 implements the correct formulas but lacks small angle approximation, causing numerical instability. AutoP2C completely fails by optimizing the pose matrix directly without manifold constraints.

Component C2: Interest Region Sampling with Dilation - All Baselines

Interest region sampling computes gradient magnitudes using Sobel filters, thresholds to create a binary mask, applies morphological dilation (5×5 kernel, 2 iterations), and samples rays from the dilated regions.

```
# \nerfify\ : Complete interest region with Sobel and dilation
def _compute_interest_mask(image_hw3, threshold, dilate_iters):
    # Sobel filters for edge detection
    kx = torch.tensor([[1, 0, -1], [2, 0, -2], [1, 0, -1]])
    ky = torch.tensor([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])
    gx = F.conv2d(gray_, kx, padding=1)
    gy = F.conv2d(gray_, ky, padding=1)
    grad = torch.sqrt(gx.pow(2) + gy.pow(2))
    mask = (grad_norm >= threshold).float()
    # CORRECT: Morphological dilation via max pooling
    for _ in range(dilate_iters):
        mask = F.max_pool2d(mask, kernel_size=5, stride=1, padding=2)
```

Listing 344. C2 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Simplified gradient without Sobel or dilation
def interest_region_sampling(image, num_rays=2048):
    # WRONG: Simple difference instead of Sobel
    grad_x = torch.abs(image[:, :, 1:] - image[:, :, :-1])
    grad_y = torch.abs(image[:, 1:, :] - image[:, :-1, :])
    grad = F.pad(grad_x[..., :-1] + grad_y[..., :-1], (0, 1, 0, 1))
    # MISSING: No morphological dilation
    prob = grad.flatten() / grad.sum()
    idx = torch.multinomial(prob, num_rays, replacement=False)
# ERROR: Missing core algorithm components
```

Listing 345. C2 Implementation: GPT-5 (Score: 0.4)

```
# DeepSeek R1: Complete with OpenCV Sobel and dilation
def interest_region_sampling(self, pose, image, num_rays):
    gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    grad_x = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=3)
    grad_y = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3)
    grad_mag = np.sqrt(grad_x**2 + grad_y**2)
    mask = (grad_mag > 0.2).astype(np.uint8)
    # CORRECT: 5x5 kernel, 2 iterations
    kernel = np.ones((5, 5), np.uint8)
    mask = cv2.dilate(mask, kernel, iterations=2)
```

Listing 346. C2 Implementation: DeepSeek R1 (Score: 1.0)

```
# Paper2Code: Comprehensive interest region implementation
def interest_region_sampling(self, observed_img, num_rays):
    sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=3)
    sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3)
    gradient_magnitude = np.sqrt(sobelx**2 + sobely**2)
    binary_mask = (gradient_magnitude >= 0.2).astype(np.uint8)
    # CORRECT: Proper dilation parameters
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5))
    dilated_mask = cv2.dilate(binary_mask, kernel, iterations=2)
```

Listing 347. C2 Implementation: Paper2Code (Score: 1.0)

```
# AutoP2C: No interest sampling at all
def generate_rays(self):
    for pose in self.camera_poses:
        directions = self.compute_ray_directions(self.image_size,
                                                 self.intrinsics['focal_length'])
        rays_world = self.transform_rays(directions, pose)
# CRITICAL ERROR: Generates rays for ALL pixels
# MISSING: No gradient computation, no sampling strategy
```

Listing 348. C2 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY , DeepSeek R1, and Paper2Code correctly implement complete interest region sampling with Sobel filters and morphological dilation. GPT-5 uses simplified gradients without proper edge detection or dilation. AutoP2C completely lacks any interest sampling mechanism.

Component C3: Interest Point Sampling - All Baselines

Interest point sampling detects corners/features and samples rays from high-gradient locations without dilation.

```
# \nerfify\ : Interest point via top-k gradient magnitudes
elif strategy == "interest_point":
    # Sample from top-k gradient magnitudes (no dilation)
    mag = torch.sqrt(gx.pow(2) + gy.pow(2))
    flat = mag.view(-1)
    k = min(N * 10, flat.numel())
    topk = torch.topk(flat, k=k, largest=True)
    idxs = topk.indices
    ys, xs = idxs // W, idxs % W
# CORRECT: Distinct from region sampling
```

Listing 349. C3 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: No separate interest point implementation
# Uses same simplified gradient approach for all sampling
def interest_region_sampling(image, num_rays=2048):
    grad = F.pad(grad_x[..., :-1] + grad_y[..., :-1])
    prob = grad.flatten() / grad.sum()
# ISSUE: No distinction between point and region sampling
```

Listing 350. C3 Implementation: GPT-5 (Score: 0.5)

```python
# DeepSeek R1: Harris corner detection for interest points
def interest_point_sampling(self, pose, observed_image, batch_size):
    gray = cv2.cvtColor(observed_image, cv2.COLOR_RGB2GRAY)
    corners = cv2.cornerHarris(gray.astype(np.float32), 2, 3, 0.04)
    corners = cv2.dilate(corners, None)  # Local maxima
    threshold = 0.01 * corners.max()
    interest_mask = corners > threshold
    interest_coords = np.argwhere(interest_mask)
# CORRECT: Proper corner detection
```

Listing 351. C3 Implementation: DeepSeek R1 (Score: 1.0)

```python
# Paper2Code: goodFeaturesToTrack for interest points
def _interest_point_sampling(self, image_np, H, W):
    corners = cv2.goodFeaturesToTrack(gray,
        maxCorners=self.batch_size, qualityLevel=0.01, minDistance=10)
    if corners is not None:
        keypoints = corners.reshape(-1, 2)
# CORRECT: Standard feature detection
```

Listing 352. C3 Implementation: Paper2Code (Score: 1.0)

```python
# AutoP2C: No interest point sampling
# MISSING: No feature detection or corner detection
# MISSING: No sampling strategy differentiation
```

Listing 353. C3 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY , DeepSeek R1, and Paper2Code implement distinct interest point sampling methods. GPT-5 doesn't differentiate between point and region sampling. AutoP2C lacks any interest-based sampling.

Component C4: Photometric RGB MSE Loss - All Baselines

The paper uses standard RGB mean squared error loss: $\mathcal{L} = \sum_{r \in \mathcal{R}} \|\hat{C}(r) - C(r)\|_2^2$.

```python
# \nerfify\ : Standard RGB MSE via Nerfacto base class
loss_dict = super().get_loss_dict(outputs, batch, metrics_dict)
# Inherits correct RGB loss: torch.mean((pred - gt)**2)
```

Listing 354. C4 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Correct RGB MSE
loss = F.mse_loss(rendered, observed)
# CORRECT: Standard photometric loss
```

Listing 355. C4 Implementation: GPT-5 (Score: 1.0)

```python
# DeepSeek R1: RGB MSE implementation
if loss_type == 'l2':
    loss = torch.mean((rendered_rgb - observed_rgb) ** 2)
# CORRECT: Standard MSE loss
```

Listing 356. C4 Implementation: DeepSeek R1 (Score: 1.0)

```python
# Paper2Code: Photometric loss with proper normalization
def photometric_loss(predicted, observed):
    difference = predicted - observed
    squared_difference = difference ** 2
    photometric_loss = total_error / (H * W * 3)
# CORRECT: Normalized MSE
```

Listing 357. C4 Implementation: Paper2Code (Score: 1.0)

```python
# AutoP2C: Generic loss without context
def photometric_loss(predicted, observed):
    difference = predicted - observed
    squared_difference = difference ** 2
# ISSUE: Oversimplified, no ray context
```

Listing 358. C4 Implementation: AutoP2C (Score: 0.5)

Analysis: All methods implement RGB MSE loss, but AutoP2C's implementation is overly generic without ray-based context.

Component C5: YUV-UV Loss Variant - All Baselines

LineMOD variant uses only U and V channels in YUV space with BT.709 conversion matrix.

```python
# \nerfify\ : YUV-UV loss with BT.709 standard
if self.config.use_yuv_uv_loss:
    yuv_gt = rgb_to_yuv_bt709(image)  # BT.709 coefficients
    yuv_pr = rgb_to_yuv_bt709(pred)
    gt_uv = yuv_gt[..., 1:]  # U and V only
    pr_uv = yuv_pr[..., 1:]
    uv_loss = torch.mean((gt_uv - pr_uv) ** 2)
# CORRECT: Exact paper specification
```

Listing 359. C5 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: No YUV loss implementation
# MISSING: Only RGB MSE, no YUV variant
```

Listing 360. C5 Implementation: GPT-5 (Score: 0.0)

```python
# DeepSeek R1: YUV with different coefficients
elif loss_type == 'yuv':
    rgb_to_yuv = torch.tensor([
        [0.299, 0.587, 0.114],  # ISSUE: BT.601, not BT.709
        [-0.14713, -0.28886, 0.436],
        [0.615, -0.51499, -0.10001]
    ])
    loss = torch.mean((rendered_yuv[:, 1:] - observed_yuv[:, 1:]) ** 2)
# ERROR: Wrong color space standard
```

Listing 361. C5 Implementation: DeepSeek R1 (Score: 0.8)

```python
# Paper2Code: Correct BT.709 YUV conversion
conversion_matrix = np.array(
    [[0.2126, 0.7152, 0.0722],     # CORRECT: BT.709
     [-0.09991, -0.33609, 0.436],
     [0.615, -0.55861, -0.05639]])
yuv_image = torch.tensordot(image, conversion_matrix.T, dims=([2], [1]))
```

Listing 362. C5 Implementation: Paper2Code (Score: 1.0)

```python
# AutoP2C: No YUV implementation
# MISSING: No color space conversion
```

Listing 363. C5 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY and Paper2Code correctly implement BT.709 YUV conversion. DeepSeek R1 uses wrong coefficients (BT.601). GPT-5 and AutoP2C lack YUV loss entirely.

Component C6: Camera LR Schedule - All Baselines

Paper specifies exponential decay: $\alpha_t = \alpha_0 \cdot 0.8^{t/100}$.

```python
# \nerfify\ : Exact formula via training callback
def set_camera_lr(step: int) -> None:
    lr = self._camera_lr0 * (0.8 ** (step / 100.0))
    for g in optim.param_groups:
        g["lr"] = lr
# CORRECT: Precise paper formula
```

Listing 364. C6 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Fixed learning rate
optimizer = torch.optim.Adam([pose], lr=lr)
# MISSING: No learning rate schedule
```

Listing 365. C6 Implementation: GPT-5 (Score: 0.0)

```python
# DeepSeek R1: Wrong exponential schedule
self.scheduler = torch.optim.lr_scheduler.
    ExponentialLR(
    self.optimizer, gamma=0.8)
# ERROR: gamma^t instead of gamma^(t/100)
```

Listing 366. C6 Implementation: DeepSeek R1 (Score: 0.5)

```python
# Paper2Code: Correct custom schedule
def get_learning_rate(initial_lr, decay_factor,
    decay_steps, iteration):
    lr = initial_lr * (decay_factor ** (iteration /
    decay_steps))
    return lr
# CORRECT: Matches paper formula
```

Listing 367. C6 Implementation: Paper2Code (Score: 1.0)

```python
# AutoP2C: No learning rate schedule
optimizer = optim.Adam([pose], lr=0.01)
# MISSING: Fixed LR throughout
```

Listing 368. C6 Implementation: AutoP2C (Score: 0.0)

Analysis: Only NERFIFY and Paper2Code implement the exact LR schedule. DeepSeek R1 uses wrong decay rate. GPT-5 and AutoP2C have no scheduling.

Component C7: Small Ray Batch Size - All Baselines

Paper uses 2048 rays per optimization step for efficiency.

```python
# \nerfify\ : Exact batch size
datamanager=TemplateDataManagerConfig(
    train_num_rays_per_batch=2048,  # CORRECT: Paper
    value
    eval_num_rays_per_batch=4096,
)
```

Listing 369. C7 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Correct batch size
def interest_region_sampling(image, num_rays=2048):
# CORRECT: 2048 rays
```

Listing 370. C7 Implementation: GPT-5 (Score: 1.0)

```python
# DeepSeek R1: Wrong default
def __init__(self, batch_size=512):  # ERROR: Default
    512
    self.batch_size = batch_size
```

Listing 371. C7 Implementation: DeepSeek R1 (Score: 0.5)

```python
# Paper2Code: Correct from config
self.batch_size = config["ray_sampling"].get("
    batch_size", 2048)
# CORRECT: 2048 default
```

Listing 372. C7 Implementation: Paper2Code (Score: 1.0)

```python
# AutoP2C: No batch control
rays_world = self.transform_rays(directions, pose)
# CRITICAL ERROR: Processes full image
```

Listing 373. C7 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY , GPT-5, and Paper2Code use correct batch size. DeepSeek R1 defaults to 512. AutoP2C processes entire images.

Component C8: Gradient-based Pose Updates - All Baselines

All methods use gradient descent, but implementation quality varies.

```python
# \nerfify\ : Proper gradient flow through volume
    rendering
loss = self.get_loss_dict(outputs, batch)
loss.backward()
optimizer.step()
# CORRECT: Full differentiable pipeline
```

Listing 374. C8 Implementation: NERFIFY (Score: 1.0)

```python
# Standard gradient descent present in all baselines
loss.backward()
optimizer.step()
```

Listing 375. C8 Implementation: All Others (Score: 0.8-1.0)

Analysis: All methods implement basic gradient descent, with varying quality in the overall pipeline.

Component C9: Volume Rendering Integration - All Baselines

Standard NeRF volume rendering with alpha compositing.

```python
# \nerfify\ : Uses Nerfacto's volume renderer
# Inherits complete volume rendering pipeline
```

Listing 376. C9 Implementation: NERFIFY (Score: 1.0)

```python
# All baselines have some volume rendering
alpha = 1.0 - torch.exp(-sigma * deltas)
weights = alpha * transmittance
final_color = torch.sum(weights * rgb, dim=1)
```

Listing 377. C9 Implementation: Others (Score: 0.6-1.0)

Analysis: All methods include volume rendering with varying completeness.

Component C10: Pose-only Optimization Mode - All Baselines

Freezing NeRF weights and only optimizing camera poses.

```
# \nerfify\ : Explicit pose-only mode
if self.config.invert_poses_only:
    for p in self.parameters():
        p.requires_grad = False
    for p in self.camera_optimizer.parameters():
        p.requires_grad = True
# CORRECT: Clear separation
```

Listing 378. C10 Implementation: NERFIFY (Score: 1.0)

```
# No explicit pose-only mode in other baselines
# MISSING: All assume fixed NeRF implicitly
```

Listing 379. C10 Implementation: Others (Score: 0.0)

Analysis: Only NERFIFY has explicit pose-only optimization control.

3.8.6. Scoring Analysis

Table 63. iNeRF Implementation Scores

Method	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	Weighted Avg_{LLM}
NERFIFY	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.7	0.97
GPT-5	0.8	0.4	0.5	1.0	0.0	0.0	1.0	1.0	0.8	0.0	0.58
DeepSeek R1	1.0	1.0	1.0	1.0	0.8	0.5	0.5	1.0	1.0	0.0	0.68
Paper2Code	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.8	0.0	0.75
AutoP2C	0.0	0.0	0.0	0.5	0.0	0.0	0.0	0.3	0.6	0.0	0.05

3.8.7. Why Baselines Fail Despite Component Scores

GPT-5 (Score: 0.58 components, 0% trainable)

- **Strengths**: Implements SE(3) structure, basic volume rendering, correct batch size
- **Fatal Issues**:
  - No Nerfstudio integration - standalone script cannot be deployed
  - Missing small angle handling causes numerical instability
  - Simplified gradient computation without proper edge detection
  - No learning rate schedule leads to poor convergence
- **Result**: Code runs but produces unstable optimization with divergent poses

DeepSeek R1 (Score: 0.68 components, 0% trainable)

- **Strengths**: Complete SE(3) implementation, proper interest sampling, YUV loss variant
- **Fatal Issues**:
  - No framework integration - requires complete rewrite for Nerfstudio
  - Wrong LR schedule causes 100× faster decay than intended
  - Default batch size mismatch reduces gradient quality
- **Result**: Mathematically sound but practically unusable without major refactoring

Paper2Code (Score: 0.75 components, 0% trainable)

- **Strengths**: Most complete component implementation, correct hyperparameters
- **Fatal Issues**:
  - 2100+ lines of framework-agnostic code incompatible with Nerfstudio
  - Over-engineered module structure prevents simple integration
  - Requires specific config.yaml format not matching Nerfstudio conventions
- **Result**: High-quality standalone system that cannot be deployed as intended plugin

AutoP2C (Score: 0.05 components, 0% trainable)

- **Strengths**: None - fundamental algorithmic failures throughout
- **Fatal Issues**:
  - No SE(3) manifold constraints - optimizes invalid rotation matrices
  - Missing all sampling strategies - processes entire images
  - Import errors prevent compilation (non-existent colmap_wrapper)
  - Empty placeholder functions throughout codebase
- **Result**: Code neither compiles nor demonstrates any domain understanding

3.8.8. Hyperparameter Fidelity

Table 64. Hyperparameter Implementation Accuracy

Parameter	Paper	NERFIFY	GPT-5	R1	P2C	AutoP2C
Ray batch size	2048	✓	✓	×	✓	×
Initial LR (α_0)	0.01	✓	×	✓	✓	✓
LR decay factor	0.8	✓	×	×	✓	×
LR decay steps	100	✓	×	×	✓	×
Optimizer	Adam	✓	✓	✓	✓	✓
Adam β_1	0.9	✓	×	✓	×	×
Adam β_2	0.999	✓	×	✓	×	×
Edge threshold	0.2	✓	×	✓	✓	×
Dilation iterations	2	✓	×	✓	✓	×
Kernel size	5×5	✓	×	✓	✓	×
W Score	–	1.00	0.20	0.60	0.80	0.10

3.8.9. Conclusion

All baselines attempt to implement iNeRF with varying sophistication. Paper2Code achieves the highest component coverage (75%) with correct SE(3) parameterization and complete interest sampling but produces framework-agnostic code. DeepSeek R1 implements core algorithms correctly (68%) but uses wrong hyperparameters that prevent convergence. GPT-5 provides a functional skeleton (58%) missing critical details like small angle handling and proper edge detection. AutoP2C completely fails (5%) with no SE(3) constraints, missing sampling strategies, and import errors. Only NERFIFY produces immediately trainable code as a complete Nerfstudio plugin with 97% component fidelity and exact hyperparameter matching, demonstrating that the 0% trainable rate for baselines versus 100% for NERFIFY validates the necessity of domain-specific synthesis for complex vision research.

3.9. SIGNeRF: Scene Integrated Generation for Neural Radiance Fields

3.9.1. Paper Overview

SIGNeRF introduces a framework for editing neural radiance fields by generating multi-view consistent reference sheets through depth-conditioned ControlNet inpainting. The method enables scene-integrated generation by first creating a reference sheet with multiple views arranged in a grid, processing them through ControlNet with depth conditioning, and then propagating the edits to all training views. This addresses the critical challenge of maintaining 3D consistency when editing NeRF scenes with diffusion models.

3.9.2. Implementation Overview

Table 65. SIGNeRF Implementation Summary Across All Baselines

Aspect	NERFIFY	GPT-5	DeepSeek R1	Paper2Code	AutoP2C
Lines of Code	482	876	624	1842	658
File Organization	Plugin	Single	Single	Multi-file	Multi-file
Reference Sheet	✓	✓	Partial	✓	×
ControlNet Integration	✓	✓	Wrong Type	✓	×
Dataset Updating	✓	Partial	Simplified	✓	×
Two-Stage Training	✓	✓	Simplified	Partial	×
Selection Modes	✓	Partial	Partial	Partial	×
Trainable	✓	×	×	×	×

Note: NERFIFY always produces trainable code as a complete Nerfstudio plugin

3.9.3. Novel Components

Table 66. Novel Components in SIGNeRF with Importance Weights

ID	Component	Weight w_i
C1	Reference sheet generation with grid layout	0.20
C2	ControlNet depth-conditioned inpainting	0.20
C3	Multi-view dataset updating	0.15
C4	Two-stage NeRF training (initial + fine-tune)	0.10
C5	Selection modes (bounding box and proxy)	0.10
C6	Iterative refinement with parameter adjustment	0.10
C7	ControlNet hyperparameters (scale, guidance)	0.08
C8	LPIPS loss for consistency	0.07

3.9.4. Quantitative Metrics

Table 67. SIGNeRF Implementation Coverage Metrics

Method	C	I	M	W	$\text{Score}_{\text{LLM}}$
NERFIFY (Ours)	1.00	0.00	0.00	1.00	1.00
GPT-5	0.50	0.25	0.25	0.63	0.58
DeepSeek R1	0.63	0.25	0.12	0.75	0.72
Paper2Code	0.38	0.38	0.24	0.50	0.52
AutoP2C	0.00	0.13	0.87	0.00	0.08

3.9.5. Component-by-Component Analysis

Component C1: Reference Sheet Generation - All Baselines

The paper specifies creating a reference sheet by arranging M views in a grid with one empty slot:

$$\mathcal{R} \leftarrow \text{ControlNet}(\bar{\mathcal{I}}_R, \bar{\mathcal{D}}_R, \bar{\mathcal{M}}_R, y) \quad (1)$$

```python
# \nerfify\ : Complete grid generation with Nerfstudio integration
class ControlNetEditor:
    def create_reference_sheet(self, rgb_images, depth_images,
                               masks, prompt, grid_size=(2, 3)):
        """Create edited reference sheet with M views in grid."""
        # Arrange images into grid with empty slot
        reference_sheet = self._arrange_grid(
            rgb_images, depth_images, masks, grid_size
        )
        # Process with ControlNet
        edited_sheet = self._run_controlnet_inpainting(
            reference_sheet['rgb_grid'],
            reference_sheet['depth_grid'],
            reference_sheet['mask_grid'], prompt
        )
        return edited_sheet
# CORRECT: Proper grid layout with empty slot
```

Listing 380. C1 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Standalone implementation
def generate_reference_sheet(color_imgs, depth_imgs,
                             mask_imgs, prompt, ncols=3):
    """Combine M views into tiled reference sheet."""
    grid_color = make_grid(colors, ncols)
    grid_depth = make_grid(depths, ncols)
    grid_mask = make_grid([cv2.merge([m,m,m])
                           for m in masks], ncols)
    gen = pipe(prompt=prompt, image=grid_color,
               control_image=grid_depth, mask_image=grid_mask)
    return gen.images[0]
# ERROR: Not integrated with Nerfstudio framework
```

Listing 381. C1 Implementation: GPT-5 (Score: 0.8)

```python
# DeepSeek R1: Simplified grid generation
def generate_reference_sheet(self, prompt, selection_mode,
                             num_reference_views=5):
    reference_cameras = self._select_reference_cameras(
        num_reference_views, selection_params)
    color_grid, depth_grid, mask_grid = [], [], []
    for cam in reference_cameras:
        color, depth = self.nerf_model.render(cam)
        mask = self._generate_mask(cam, selection_mode,
                                   selection_params, depth)
        color_grid.append(color)
    grid_size = int(np.ceil(np.sqrt(num_reference_views)))
    # ISSUE: Missing actual ControlNet processing
    reference_sheet = self._create_reference_sheet(
        color_grid, depth_grid, mask_grid, grid_size, prompt)
```

Listing 382. C1 Implementation: DeepSeek R1 (Score: 0.7)

```python
# Paper2Code: Complete but standalone
def generate_reference_sheet(self, image_grid, depth_grid,
                             mask_grid, prompt=None):
    """Generate reference sheet using diffusion pipeline."""
    image_pil = self._np_to_pil(image_grid)
    depth_pil = self._np_to_pil(depth_grid)
    mask_pil = self._np_to_pil(mask_grid)
    generator = torch.Generator(device=self.device).manual_seed(
        self.seed)
    result = self.pipeline(prompt=prompt, image=image_pil,
                           mask_image=mask_pil,

    controlnet_conditioning_image=depth_pil)
    return self._pil_to_np(result.images[0])
# ERROR: Not a Nerfstudio plugin
```

Listing 383. C1 Implementation: Paper2Code (Score: 1.0)

```python
# AutoP2C: No reference sheet generation
class BlenderDataset:
    def load_data(self):
        """Loads images and camera poses."""
        with open('transforms.json', 'r') as f:
            metadata = json.load(f)
            self.camera_poses = [np.array(frame['transform_matrix'])
                                 for frame in metadata['frames']]
# MISSING: No reference sheet or ControlNet implementation
```

Listing 384. C1 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY and Paper2Code correctly implement grid-based reference sheet generation, but only NERFIFY integrates it with Nerfstudio. GPT-5 has correct logic but lacks framework integration. DeepSeek R1 has the structure but misses actual ControlNet processing. AutoP2C completely lacks this component.

Component C2: ControlNet Depth-Conditioned Inpainting - All Baselines

The paper specifies using SDXL with depth ControlNet for multi-view consistency.

```python
# \nerfify\ : Correct depth ControlNet setup
from diffusers import (
    StableDiffusionXLControlNetInpaintPipeline,
                        ControlNetModel)
self.controlnet_editor = ControlNetEditor(
    controlnet_scale=self.config.controlnet_scale,
      # 0.7
    guidance_scale=self.config.guidance_scale,
      # 7.5
    denoising_strength=self.config.denoising_strength
      # 0.8
)
# CORRECT: Uses depth ControlNet as specified
```

Listing 385. C2 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Correct depth ControlNet
controlnet = ControlNetModel.from_pretrained(
    "diffusers/controlnet-depth-sdxl-1.0",
    torch_dtype=torch.float16)
self.pipe =
    StableDiffusionXLControlNetInpaintPipeline\
    .from_pretrained("stabilityai/stable-diffusion-xl-base-1.0",
                     controlnet=controlnet,
                     torch_dtype=torch.float16).to(device)
# CORRECT: Proper depth conditioning
```

Listing 386. C2 Implementation: GPT-5 (Score: 1.0)

```python
# DeepSeek R1: Wrong ControlNet type
if self.config['use_sdxl']:
    controlnet = ControlNetModel.from_pretrained(
        "diffusers/controlnet-canny-sdxl-1.0",  # WRONG!
        torch_dtype=torch.float16)
# CRITICAL ERROR: Uses Canny instead of Depth
```

Listing 387. C2 Implementation: DeepSeek R1 (Score: 0.5)

```python
# Paper2Code: Correct but hardcoded parameters
self.pipeline =
    StableDiffusionControlNetInpaintPipeline\
    .from_pretrained("stabilityai/stable-diffusion-xl-base-1.0",
                     torch_dtype=torch.float16,
    revision="fp16")
self.controlnet_scale = float(
    diffusion_config.get("controlnet_scale", [0.4, 1.0])[0]
)  # Uses 0.4 instead of paper's 0.7-0.8
# ISSUE: Wrong hyperparameters
```

Listing 388. C2 Implementation: Paper2Code (Score: 0.9)

```python
# AutoP2C: No ControlNet implementation
class ControlNet(nn.Module):
    def __init__(self, latent_dim=512, control_scale=0.7):
        super(ControlNet, self).__init__()
        self.encoder = self.create_encoder()
        # Just a placeholder NN module
# MISSING: No actual diffusion model integration
```

Listing 389. C2 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY and GPT-5 correctly use depth ControlNet. DeepSeek R1 critically uses the wrong ControlNet type (Canny), breaking 3D consistency. Paper2Code has correct implementation but wrong hyperparameters. AutoP2C has no real ControlNet.

Component C3: Multi-View Dataset Updating - All Baselines

The paper describes propagating edits to all views by compositing each into the reference sheet.

```python
# \nerfify\ : Complete view propagation
def propagate_to_views(self, reference_sheet, view_rgbs,
                       view_depths, view_masks, prompt):
    edited_views = []
    for rgb, depth, mask in zip(view_rgbs, view_depths, view_masks):
        composite_sheet = self._replace_grid_slot(
            reference_sheet, rgb, depth, mask, slot_index=-1)
        edited_view = self._run_controlnet_inpainting(
            composite_sheet['rgb_grid'],
            composite_sheet['depth_grid'],
            composite_sheet['mask_grid'], prompt)
        edited_slot = self._extract_grid_slot(edited_view,
```

```
        slot_index=-1)
            edited_views.append(edited_slot)
        return edited_views
# CORRECT: Proper view compositing and extraction
```

Listing 390. C3 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Dataset update but saves to disk
def update_dataset(ref_sheet_path, color_imgs, depth_imgs,
                   mask_imgs, prompt, out_dir):
    ref_sheet = Image.open(ref_sheet_path).convert("RGB")
    for idx, (ci, di, mi) in enumerate(zip(color_imgs,
                                           depth_imgs, mask_imgs)):
        grid = make_grid([np.array(ref_sheet), np.array(color)],
                         ncols=2)
        gen = pipe(prompt=prompt, image=grid, control_image=depth,
                   mask_image=mask).images[0]
        gen.save(os.path.join(out_dir, f"view_{idx:04d}.png"))
# ERROR: Saves to disk instead of returning for training
```

Listing 391. C3 Implementation: GPT-5 (Score: 0.7)

```
# DeepSeek R1: Incomplete update logic
def update_image_set(self, reference_sheet):
    for i, cam in self.camera_poses.items():
        color, depth = self.nerf_model.render(cam)
        mask = self._get_camera_mask(cam)
        # MISSING: No actual ControlNet propagation
        edited_view = reference_sheet  # WRONG!
        self.edited_images[i] = edited_view
```

Listing 392. C3 Implementation: DeepSeek R1 (Score: 0.4)

```
# Paper2Code: Complete implementation
def update_view(self, reference_sheet, view_rgb, view_depth,
                view_mask, prompt):
    composite_rgb = self._inject_view_into_grid(
        self.ref_sheet, view_rgb, self.empty_slot_position)
    result = self.pipeline(prompt=prompt, image=composite_rgb_pil,
                           mask_image=composite_mask_pil,

                           controlnet_conditioning_image=composite_depth_pil)
    updated_view = updated_composite_np[y_start:y_end,
                                        x_start:x_end, :]
    return updated_view
# ERROR: Not integrated with Nerfstudio training
```

Listing 393. C3 Implementation: Paper2Code (Score: 1.0)

```
# AutoP2C: No dataset updating
def update_dataset(ref_sheet_path, view_list, prompt):
    # Function defined but no implementation
    pass
# MISSING: No implementation at all
```

Listing 394. C3 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY and Paper2Code correctly implement view propagation with compositing. GPT-5 has the logic but saves to disk. DeepSeek R1 lacks actual ControlNet propagation. AutoP2C has no implementation.

Component C4: Two-Stage NeRF Training - All Baselines

The paper specifies training initial NeRF then fine-tuning on edited data.

```
# \nerfify\ : Integrated two-stage with Nerfstudio
class SIGNeRFPipeline(VanillaPipeline):
    def get_train_loss_dict(self, step: int):
        ray_bundle, batch = self.datamanager.next_train(step)
        model_outputs = self.model(ray_bundle)
        loss_dict = self.model.get_loss_dict(
            model_outputs, batch, metrics_dict)
        return model_outputs, loss_dict, metrics_dict
# Config: initial_training: 30k iterations
#         fine_tuning: load checkpoint, continue
# CORRECT: Proper two-stage pipeline
```

Listing 395. C4 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Explicit two stages but standalone
def train(self, data_dir, output_name="nerf", steps=30000):
    config = TrainerConfig(method_name="nerfacto",
                           max_num_iterations=steps,
                           data=data_dir)
    trainer = Trainer(config)
    trainer.train()

def finetune(self, pretrained_dir, edited_data_dir, steps=10000):
    config = TrainerConfig(load_dir=pretrained_dir,
                           max_num_iterations=steps,
                           data=edited_data_dir)
# ERROR: Not a Nerfstudio plugin
```

Listing 396. C4 Implementation: GPT-5 (Score: 0.9)

```
# DeepSeek R1: Simplified placeholder
def train_nerf(self, images, camera_poses):
    self.nerf_model.train_model(images, camera_poses)

def refine_nerf(self, num_iterations=1000):
    self.nerf_model.fine_tune(edited_images_list,
                              camera_poses_list, num_iterations)
# ISSUE: SimplifiedNeRF is dummy implementation
```

Listing 397. C4 Implementation: DeepSeek R1 (Score: 0.6)

```
# Paper2Code: Generic training only
class NeRFTrainer:
    def train(self):
        for iter_idx in range(1, self.iterations + 1):
            batch = self.training_data[iter_idx % num_batches]
            prediction = self.model(batch)
            loss = F.mse_loss(prediction, batch)
# MISSING: No fine-tuning stage
```

Listing 398. C4 Implementation: Paper2Code (Score: 0.4)

```
# AutoP2C: Generic NeRF training
class Trainer:
    def train(self):
        for epoch in range(self.num_epochs):
```

```
            predictions = self.model(rays['coords'],
      rays['view_dirs'])
            loss = self.compute_loss(predictions,
      batch['images'])
# MISSING: No two-stage process at all
```

Listing 399. C4 Implementation: AutoP2C (Score: 0.0)

Analysis: Only NERFIFY properly integrates two-stage training with Nerfstudio. GPT-5 has explicit stages but standalone. DeepSeek R1 has placeholder implementation. Paper2Code and AutoP2C lack fine-tuning.

Component C5: Selection Modes - All Baselines

The paper specifies bounding box and proxy mesh selection modes.

```
# \nerfify\ : Complete selection modes
def generate_mask(self, depth_map, selection):
    mode = selection.get("mode", self.default_mode)
    if mode == "bounding_box":
        bbox = selection["bbox"]
        region = depth_map[bbox["y_min"]:bbox["y_max"
    ],
                           bbox["x_min"]:bbox["x_max"
    ]]
        mask = (depth_map >= th_min) & (depth_map <=
    th_max)
    elif mode == "proxy":
        proxy_mask = selection.get("proxy_mask")
        proxy_depth = selection.get("proxy_depth")
        # Full implementation for both options
# CORRECT: Both modes fully implemented
```

Listing 400. C5 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Basic selection modes
def generate_masks_from_bbox(depths, bbox):
    x1,y1,x2,y2 = bbox
    masks=[]
    for d in depths:
        m=np.zeros_like(d,dtype=np.uint8)
        m[y1:y2,x1:x2]=255
        masks.append(m)
def proxy_mask_from_mesh(depths, proxy_depths):
    masks.append(((p>0)&(p<d)).astype(np.uint8)*255)
# ISSUE: Simplified, not integrated
```

Listing 401. C5 Implementation: GPT-5 (Score: 0.7)

```
# DeepSeek R1: Both modes but simplified
def _shape_selection_mask(self, camera,
    selection_params,
                          depth_map, h, w):
    bbox = selection_params['bounding_box']
    # Project bounding box to image space
def _proxy_selection_mask(self, camera,
    selection_params,
                          depth_map, h, w):
    proxy_mesh = selection_params['proxy_mesh']
    # Render proxy mesh from camera view
# ISSUE: Incomplete implementations
```

Listing 402. C5 Implementation: DeepSeek R1 (Score: 0.7)

```
# Paper2Code: Only bounding box mode
if mode == "bounding_box":
    bbox = selection["bbox"]
    region = depth_map[y_min:y_max, x_min:x_max]
    mask_bool = (depth_map >= th_min) & (depth_map <=
      th_max)
# MISSING: No proxy mode implementation
```

Listing 403. C5 Implementation: Paper2Code (Score: 0.3)

```
# AutoP2C: No selection modes
# No implementation found
```

Listing 404. C5 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY fully implements both selection modes. GPT-5 and DeepSeek R1 have both but simplified. Paper2Code only has bounding box. AutoP2C has none.

Component C6: Iterative Refinement - All Baselines

The paper describes iterative refinement with parameter adjustment.

```
# \nerfify\ : Full iterative refinement support
# Config allows multiple iterations:
trainer_config = TrainerConfig(
    pipeline=SIGNeRFPipelineConfig(
        model=SIGNeRFModelConfig(
            use_lpips_loss=True,
            lpips_loss_mult=0.01,
            depth_supervision_mult=0.1
        )
    )
)
# CORRECT: Supports iterative refinement
```

Listing 405. C6 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Optional second iteration
if second_iter:
    self.ref_gen.controlnet_scale=1.0
    self.ref_gen.denoising_strength=0.5
    ref_sheet2=os.path.join(base_dir,"
     reference_sheet_iter2.png")
    self.ref_gen.generate_reference_sheet(...)
    self.trainer.finetune(...)
# ISSUE: Only supports one refinement
```

Listing 406. C6 Implementation: GPT-5 (Score: 0.6)

```
# DeepSeek R1: Iterative refinement method
def iterative_refinement(self, num_iterations=2):
    for iteration in range(num_iterations):
        if iteration > 0:
            self.config['controlnet_scale'] = min
     (1.0,
                self.config.get('controlnet_scale',
     0.8) + 0.2)
        reference_sheet = self.
     generate_reference_sheet(...)
        self.update_image_set(reference_sheet)
        self.refine_nerf(500)
# CORRECT: Good iterative structure
```

Listing 407. C6 Implementation: DeepSeek R1 (Score: 0.8)

```
# Paper2Code: No iterative refinement
# Single pass only, no iteration support
```

Listing 408. C6 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: No iterative refinement
# No implementation found
```

Listing 409. C6 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY supports full iterative refinement through config. DeepSeek R1 has good iterative structure. GPT-5 only supports one refinement. Paper2Code and AutoP2C lack this feature.

Component C7: ControlNet Hyperparameters - All Baselines

The paper specifies controlnet$_scale$0.4 − 1.0, $guidance_scale$6 − 10, $denoising_strength$0.5 − 0.95.

```
# \nerfify\ : Correct hyperparameters
self.controlnet_editor = ControlNetEditor(
    controlnet_scale=self.config.controlnet_scale,
      # 0.7
    guidance_scale=self.config.guidance_scale,
      # 7.5
    denoising_strength=self.config.denoising_strength
      # 0.8
)
# CORRECT: All within paper ranges
```

Listing 410. C7 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Correct parameters
self.controlnet_scale = controlnet_scale  # 0.8
self.guidance_scale = guidance_scale      # 7.0
self.denoising_strength = denoising_strength # 0.8
# CORRECT: All within specified ranges
```

Listing 411. C7 Implementation: GPT-5 (Score: 1.0)

```
# DeepSeek R1: Mostly correct
controlnet_conditioning_scale=self.config.get('
    controlnet_scale', 0.8)
guidance_scale=self.config.get('guidance_scale', 7.5)
num_inference_steps=15  # Too few, paper uses 40-50
# ISSUE: Inference steps too low
```

Listing 412. C7 Implementation: DeepSeek R1 (Score: 0.8)

```
# Paper2Code: Wrong values from config
self.controlnet_scale = float(controlnet_scale_range
    [0])  # 0.4
self.guidance_scale = float(guidance_scale_range[0])
          # 6.0
self.denoising_strength = float(denoising_range[0])
          # 0.5
# ERROR: Takes first value from ranges, suboptimal
```

Listing 413. C7 Implementation: Paper2Code (Score: 0.5)

```
# AutoP2C: No ControlNet parameters
control_scale = 0.7  # Defined but never used
# MISSING: No actual ControlNet usage
```

Listing 414. C7 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY and GPT-5 use correct hyperparameters. DeepSeek R1 mostly correct but inference steps too low. Paper2Code uses suboptimal values. AutoP2C has no real parameters.

Component C8: LPIPS Loss for Consistency - All Baselines

The paper specifies using LPIPS loss for enhanced texture consistency during fine-tuning.

```
# \nerfify\ : LPIPS loss integrated
from torchmetrics.image.lpip import
    LearnedPerceptualImagePatchSimilarity
if self.config.use_lpips_loss:
    self.lpips =
     LearnedPerceptualImagePatchSimilarity(normalize=
     True)
    lpips_loss = self.lpips(pred_rgb_permuted,
     gt_rgb_permuted)
    loss_dict["lpips_loss"] = self.config.
     lpips_loss_mult * lpips_loss
# CORRECT: Proper LPIPS integration
```

Listing 415. C8 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: No LPIPS loss
# Uses standard MSE loss only
# MISSING: No LPIPS implementation
```

Listing 416. C8 Implementation: GPT-5 (Score: 0.0)

```
# DeepSeek R1: No LPIPS loss
# Standard NeRF losses only
# MISSING: No LPIPS implementation
```

Listing 417. C8 Implementation: DeepSeek R1 (Score: 0.0)

```
# Paper2Code: No LPIPS in training
# Only mentions LPIPS in evaluation metrics
# MISSING: Not used in training loss
```

Listing 418. C8 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: LPIPS defined but misused
def lpips_loss(predictions, targets, lpips_model):
    lpips_value = lpips_model(predictions, targets)
    return lpips_value.mean()
# ERROR: Defined but never integrated in training
```

Listing 419. C8 Implementation: AutoP2C (Score: 0.5)

Analysis: Only NERFIFY properly integrates LPIPS loss in training. AutoP2C defines it but doesn't use it. Others completely lack LPIPS implementation.

3.9.6. Scoring Analysis

Table 68. SIGNeRF Implementation Scores

Method	C1	C2	C3	C4	C5	C6	C7	C8	Weighted Avg$_{LLM}$
NERFIFY	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.00
GPT-5	0.8	1.0	0.7	0.9	0.7	0.6	1.0	0.0	0.71
DeepSeek R1	0.7	0.5	0.4	0.6	0.7	0.8	0.8	0.0	0.56
Paper2Code	1.0	0.9	1.0	0.4	0.3	0.0	0.5	0.0	0.51
AutoP2C	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.06

3.9.7. Why Baselines Fail Despite Component Scores

GPT-5 (Score: 0.58 overall, 0% trainable)

- **Strengths**: Correct depth ControlNet, proper hyperparameters, explicit two-stage training
- **Fatal Issues**:
  - No Nerfstudio integration - standalone script only
  - Saves edited images to disk instead of training pipeline
  - Missing LPIPS loss for consistency
- **Result**: Cannot be loaded as 'ns-train signerf', requires 3-5 hours manual integration

DeepSeek R1 (Score: 0.72 overall, 0% trainable)

- **Strengths**: Good iterative refinement structure, both selection modes

- **Fatal Issues**:
  - Uses Canny ControlNet instead of Depth - fundamentally breaks 3D consistency
  - SimplifiedNeRF is placeholder with no real implementation
  - Missing actual ControlNet propagation in dataset update
- **Result**: Wrong ControlNet type makes method unusable for 3D-consistent editing

Paper2Code (Score: 0.52 overall, 0% trainable)

- **Strengths**: Complete reference sheet generation, correct depth ControlNet, proper view updating
- **Fatal Issues**:
  - Not a Nerfstudio plugin - multi-file standalone code
  - Missing fine-tuning stage in training
  - Wrong hyperparameter values from config
- **Result**: Core algorithm correct but cannot integrate with Nerfstudio framework

AutoP2C (Score: 0.08 overall, 0% trainable)

- **Strengths**: Basic dataset loading structure
- **Fatal Issues**:
  - No reference sheet generation or ControlNet implementation
  - No dataset updating or editing pipeline
  - Generic NeRF code without any SIGNeRF-specific features
- **Result**: Complete failure to understand paper, produces generic NeRF only

3.9.8. Hyperparameter Fidelity

Table 69. Hyperparameter Implementation Accuracy

Parameter	Paper	NERFIFY	GPT-5	R1	P2C	AutoP2C
ControlNet scale	0.4-1.0	✓	✓	✓	×	×
Guidance scale	6-10	✓	✓	✓	✓	×
Denoising strength	0.5-0.95	✓	✓	✓	✓	×
Inference steps	40-50	✓	✓	×	✓	×
Initial iterations	30000	✓	✓	×	×	×
Fine-tune iterations	10000	✓	✓	×	×	×
LPIPS weight	0.01	✓	×	×	×	×
W Score	–	1.00	0.71	0.43	0.43	0.00

3.9.9. Conclusion

All baselines attempt to implement SIGNeRF with varying levels of sophistication. GPT-5 correctly implements the core algorithm including depth ControlNet and two-stage training but lacks Nerfstudio integration. DeepSeek R1 has good structure and iterative refinement but critically uses the wrong ControlNet type (Canny instead of Depth). Paper2Code implements reference sheet generation and view updating correctly but misses fine-tuning and uses wrong hyperparameters. AutoP2C completely fails to understand the paper, producing only generic NeRF code without any editing capabilities. Only NERFIFY produces immediately trainable code.

3.10. MCNeRF: Monte Carlo Rendering and Denoising for Real-Time NeRFs

3.10.1. Paper Overview

MCNeRF [8] introduces a general-purpose acceleration technique for volumetric NeRF rendering through Monte Carlo importance sampling. The method replaces dense ray marching with stochastic sampling based on transmittance-weighted probability distributions, achieving 7× reduction in color MLP evaluations while maintaining visual quality through a lightweight image-space denoiser. This work addresses the critical bottleneck of real-time NeRF rendering by demonstrating that careful importance sampling with only 5 samples per pixel can match the quality of traditional 128-sample quadrature.

3.10.2. Implementation Overview

Table 70. MCNeRF Implementation Summary Across All Baselines

Aspect	NERFIFY	GPT-5	DeepSeek R1	Paper2Code	AutoP2C
Lines of Code	476	287	892	1243	892
File Organization	Plugin	Single	Single	Multi-file	Multi-file
Monte Carlo Sampling	✓	✓	Partial	×	×
Two-Pass Rendering	✓	✓	Partial	×	×
Importance Sampling	✓	✓	✓	×	×
Denoiser Network	✓	✓	✓	Simplified	Attempted
Trainable	✓	×	×	×	×

Note: NERFIFY always produces trainable code as a complete Nerfstudio plugin

3.10.3. Novel Components

Table 71. Novel Components in MCNeRF with Importance Weights

ID	Component	Weight w_i
C1	Monte Carlo integration replacing dense quadrature	0.25
C2	Importance sampling via transmittance CDF	0.20
C3	Stratified sampling for variance reduction	0.15
C4	Two-pass rendering (density then color)	0.15
C5	Lightweight 3-layer CNN denoiser	0.10
C6	Combined MSE + SSIM loss	0.05
C7	Per-scene denoiser training	0.05
C8	Real-time performance (5 spp target)	0.05

3.10.4. Quantitative Metrics

Table 72. MCNeRF Implementation Coverage Metrics

Method	C	I	M	W	$\text{Score}_{\text{LLM}}$
NERFIFY (Ours)	1.00	0.00	0.00	1.00	0.95
GPT-5	0.75	0.25	0.00	0.85	0.95
DeepSeek R1	0.50	0.38	0.13	0.80	0.74
Paper2Code	0.00	0.13	0.88	0.20	0.15
AutoP2C	0.00	0.25	0.75	0.10	0.08

3.10.5. Component-by-Component Analysis

Component C1: Monte Carlo Integration - All Baselines

The paper replaces standard quadrature $\hat{C}(\mathbf{r}) = \sum_{i=1}^{N} T_i \alpha_i \mathbf{c}_i$ with Monte Carlo estimation: $\hat{C}(\mathbf{r}) = \mathbb{E}_{i \sim p_i}[\mathbf{c}_i W]$ where $p_i = w_i / W$ and $W = \sum_i T_i \alpha_i$.

```
# \nerfify\ : Unbiased MC estimator with proper normalization
rgb_sel = field_outputs_sel[FieldHeadNames.RGB]  # [R,M,3]
# Unbiased estimator: mean_j( c_{i_j} ) * W
rgb_est = rgb_sel.mean(dim=1) * W  # [R,3]
# Composite with background using (1 - W)
bg = self._background_color(R, device)
rgb = rgb_est + (1.0 - W_clamped) * bg  # [R,3]
# CORRECT: Proper MC formula W * mean(c_samples)
```

Listing 420. C1 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Correct MC integration with weight normalization
for ep in tqdm(range(epochs)):
    rays_o, rays_d = rays
    preds = monte_carlo_render(rays_o, rays_d, model)
    # Proper MC estimator in monte_carlo_render
    rgb_out = rgb_out / n_samples
    return rgb_out
# CORRECT: Proper averaging over MC samples
```

Listing 421. C1 Implementation: GPT-5 (Score: 1.0)

```
# DeepSeek R1: MC sampling but simplified averaging
noisy_colors = torch.mean(colors, dim=1)  # (B, 3)
alpha = torch.mean(torch.sigmoid(densities), dim=1, keepdim=True)
# ISSUE: Missing proper weight normalization
```

Listing 422. C1 Implementation: DeepSeek R1 (Score: 0.8)

```
# Paper2Code: No MC implementation, only metrics
def calculate_psnr(predictions, ground_truth):
    mse = F.mse_loss(predictions, ground_truth)
    psnr = 20 * torch.log10(max_pixel_value) - 10 * torch.log10(mse)
# MISSING: No Monte Carlo sampling implementation
```

Listing 423. C1 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: Dataset loading only, no rendering
def generate_rays(self):
    for pose in self.camera_poses:
        directions = np.stack([(i - cx) / fx, (j - cy) / fy])
# MISSING: No MC integration, only ray generation
```

Listing 424. C1 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY and GPT-5 correctly implement the unbiased MC estimator with $W \cdot \text{mean}(c)$. DeepSeek R1 attempts MC but lacks proper weight normalization. Paper2Code and AutoP2C completely miss the MC algorithm.

Component C2: Importance Sampling via Transmittance CDF - All Baselines

The paper samples according to $p_i = w_i / W$ using inverse CDF transform on cumulative weights.

```
# \nerfify\ : Complete CDF-based importance sampling
W = weights.sum(dim=1)  # [R,1]
# CDF over samples
cdf = torch.cumsum(weights[..., 0], dim=1)  # [R,S]
# searchsorted returns indices in [0,S]
idx = torch.searchsorted(cdf, thresholds, right=True).clamp(max=S - 1)
# CORRECT: Proper inverse CDF sampling
```

Listing 425. C2 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Proper cumulative weight building
W = weights.sum(dim=-1, keepdim=True)
W[W == 0] = 1e-6
pdf = weights / W
cdfs = torch.cumsum(pdf, dim=-1)
idx = torch.searchsorted(cdfs, us, right=True).clamp(max=n_coarse - 1)
# CORRECT: Textbook inverse CDF implementation
```

Listing 426. C2 Implementation: GPT-5 (Score: 1.0)

```
# DeepSeek R1: CDF sampling with minor issues
cdf = torch.cumsum(probs, dim=-1)
indices = torch.searchsorted(cdf, u, right=True)
sampled_depths = depth_below + t * (depth_above - depth_below)
# ISSUE: Overcomplicated interpolation logic
```

Listing 427. C2 Implementation: DeepSeek R1 (Score: 0.8)

```
# Paper2Code: No importance sampling
def render_view(self, camera_pose, image_size=[800, 800]):
    rays = self.generate_rays(camera_pose, image_size)
    rendered_image = self.model.render(rays)
# MISSING: No CDF-based sampling
```

Listing 428. C2 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: Basic ray generation only
ray_directions = directions @ pose_matrix[:3, :3].T
ray_origins = np.broadcast_to(pose_matrix[:3, 3], ray_directions.shape)
# MISSING: No importance sampling logic
```

Listing 429. C2 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY and GPT-5 correctly implement inverse CDF sampling. DeepSeek R1 has the right idea but overcomplicates interpolation. Paper2Code and AutoP2C lack any importance sampling.

Component C3: Stratified Sampling for Variance Reduction - All Baselines

The paper uses stratified random sampling: $t_k = \frac{k+u_k}{M} \cdot W$ where $u_k \sim U(0,1)$.

```
# \nerfify\ : Perfect stratified sampling
if self.config.mc_stratified and M > 1:
    u = torch.rand(R, M, device=device)
    k = torch.arange(M, device=device)[None, :].expand(R, -1).float()
    thresholds = ((k + u) / float(M)) * W  # [R,M]
# CORRECT: Proper jittered uniform sampling
```

Listing 430. C3 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Correct stratification
us = torch.rand((N_rays, n_samples), device=device)
# Stratified MC sampling
cdfs = torch.cumsum(pdf, dim=-1)
idx = torch.searchsorted(cdfs, us, right=True).clamp(
    max=n_coarse - 1)
# CORRECT: Proper stratified sampling implementation
```

Listing 431. C3 Implementation: GPT-5 (Score: 1.0)

```python
# DeepSeek R1: Partial stratification
u = torch.rand(B, num_samples, device=depths.device)
u = (u + torch.arange(num_samples).float().view(1,
    -1)) / num_samples
# ISSUE: Missing device specification for arange
```

Listing 432. C3 Implementation: DeepSeek R1 (Score: 0.6)

```python
# Paper2Code: No stratified sampling
# Placeholder for ray generation logic
return []
# MISSING: No stratification implementation
```

Listing 433. C3 Implementation: Paper2Code (Score: 0.0)

```python
# AutoP2C: Uniform sampling only
i, j = np.meshgrid(np.arange(w), np.arange(h),
    indexing='xy')
# MISSING: No stratified sampling
```

Listing 434. C3 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY and GPT-5 correctly implement stratified sampling with jittered uniforms. DeepSeek R1 partially implements but has device handling issues. Paper2Code and AutoP2C lack stratification.

Component C4: Two-Pass Rendering - All Baselines

The paper specifies Pass 1 for density-only queries to build weights, Pass 2 for selective color evaluation.

```python
# \nerfify\ : Clean two-pass separation
# First pass: densities, weights, accumulations
density, density_embedding = self.field.get_density(
    ray_samples)
weights = ray_samples.get_weights(density)
# Second pass: MC sample color indices
idx = torch.searchsorted(cdf, thresholds, right=True)
field_outputs_sel = self.field.get_outputs(
    selected_ray_samples, de_sel)
# CORRECT: Clear separation of passes
```

Listing 435. C4 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Explicit two-pass with comments
# First pass: compute opacity weights
for i in range(n_coarse):
    _, sigma = model(pts, rays_d)
    weights[:, i] = T * alpha
# Second pass: evaluate radiance at sampled indices
for j in range(n_samples):
    rgb, _ = model(pts, rays_d)
# CORRECT: Well-documented two-pass structure
```

Listing 436. C4 Implementation: GPT-5 (Score: 1.0)

```python
# DeepSeek R1: Two-pass but inefficient
with torch.no_grad():
    coarse_densities = scene_representation.
    compute_density(points_flat)
# Later: compute radiance
colors_flat = scene_representation.compute_radiance(
    points_flat, rays_d_flat)
# ISSUE: Not clearly separated passes
```

Listing 437. C4 Implementation: DeepSeek R1 (Score: 0.8)

```python
# Paper2Code: Single-pass rendering assumption
rendered_image = self.model.render(rays)
# MISSING: No two-pass implementation
```

Listing 438. C4 Implementation: Paper2Code (Score: 0.0)

```python
# AutoP2C: No rendering implementation
self.rays.append((ray_origins, ray_directions))
# MISSING: No pass structure at all
```

Listing 439. C4 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY and GPT-5 correctly separate density and color evaluation. DeepSeek R1 has two passes but lacks clear separation. Paper2Code and AutoP2C miss the two-pass structure entirely.

Component C5: Lightweight 3-Layer CNN Denoiser - All Baselines

The paper specifies a 3-layer CNN with 8 output channels, ReLU activations, and 4 input channels (RGB+α).

```python
# \nerfify\ : Complete configurable denoiser
class SimpleDenoiser(nn.Module):
    def __init__(self, in_ch: int = 4, feat: int = 8)
    :
        self.net = nn.Sequential(
            nn.Conv2d(in_ch, feat, 3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(feat, feat, 3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(feat, 3, 3, padding=1),
        )
# CORRECT: Exact paper specification
```

Listing 440. C5 Implementation: NERFIFY (Score: 1.0)

```python
# GPT-5: Proper 3-layer denoiser
class DenoiseNet(nn.Module):
    def __init__(self):
        self.fc1 = nn.Linear(3, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.sigma_out = nn.Linear(hidden_dim, 1)
# CORRECT: 3-layer architecture with proper channels
```

Listing 441. C5 Implementation: GPT-5 (Score: 1.0)

```python
# DeepSeek R1: Denoiser with feature computation
self.net = nn.Sequential(
    nn.Conv2d(4, 8, kernel_size=3, padding=1),
    nn.ReLU(), nn.Conv2d(8, 8, kernel_size=3, padding
    =1),
    nn.ReLU(), nn.Conv2d(8, self.feature_dim + 2,
    kernel_size=3, padding=1))
# ISSUE: Extra complexity with feature branches
```

Listing 442. C5 Implementation: DeepSeek R1 (Score: 0.8)

```python
# Paper2Code: Simplified denoiser attempt
class DenoiseNet(nn.Module):
    def forward(self, noisy_image, alpha_channel):
        x = self.activation(self.conv1(input_tensor))
        x = self.activation(self.conv2(x))
```

```
# ERROR: Missing proper initialization
```

Listing 443. C5 Implementation: Paper2Code (Score: 0.4)

```
# AutoP2C: Placeholder denoiser
denoised_output = F.conv2d(noisy_image, x, padding=1)
# CRITICAL ERROR: No actual network defined
```

Listing 444. C5 Implementation: AutoP2C (Score: 0.2)

Analysis: NERFIFY and GPT-5 implement the exact 3-layer specification. DeepSeek R1 adds unnecessary complexity. Paper2Code has structure but missing initialization. AutoP2C only has placeholder code.

Component C6: Combined MSE + SSIM Loss - All Baselines

The paper uses $\mathcal{L} = \mathcal{L}_{\text{MSE}} + \lambda \cdot \mathcal{L}_{\text{SSIM}}$ with $\lambda = 0.1$.

```
# \nerfify\ : Proper combined loss
total_loss = mse_loss + 0.1 * ssim_loss  # lambda =
    0.1 as in paper
# CORRECT: Exact paper specification
```

Listing 445. C6 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Correct SSIM implementation
loss_recon = F.mse_loss(denoised, gt_colors)
loss_ssim = ssim_loss(denoised, gt_colors)
loss = loss_recon + 0.1 * loss_ssim
# CORRECT: Proper weighting with lambda=0.1
```

Listing 446. C6 Implementation: GPT-5 (Score: 1.0)

```
# DeepSeek R1: SSIM but simplified
ssim_loss = 1 - self._ssim(denoised_images,
    target_images)
total_loss = mse_loss + 0.1 * ssim_loss
# ISSUE: Simplified SSIM computation
```

Listing 447. C6 Implementation: DeepSeek R1 (Score: 0.6)

```
# Paper2Code: Has SSIM but no MC
ssim_val = compare_ssim(ground_truth, prediction,
    multichannel=True)
# CORRECT: SSIM computation but missing integration
```

Listing 448. C6 Implementation: Paper2Code (Score: 1.0)

```
# AutoP2C: No loss implementation
# MISSING: No loss functions defined
```

Listing 449. C6 Implementation: AutoP2C (Score: 0.0)

Analysis: NERFIFY and GPT-5 implement the exact loss combination. DeepSeek R1 has correct weighting but simplified SSIM. Paper2Code has SSIM but not integrated. AutoP2C missing entirely.

Component C7: Per-Scene Denoiser Training - All Baselines

The paper trains a separate denoiser for each scene with 100k steps.

```
# \nerfify\ : Full per-scene pipeline
# Per scene, render training views with MC sampling
# Optimize DenoiseNet with Adam, 100k steps, batch
    size 32
self.denoiser_optimizer = torch.optim.Adam(
    model.denoiser.parameters(), lr=config.get('
    denoiser_lr', 1e-3))
# CORRECT: Complete per-scene training
```

Listing 450. C7 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Per-scene training present
def train_mcnerf(model, denoiser, optimizer, rays,
    gt_colors, epochs=200):
    for ep in tqdm(range(epochs)):
        denoised = denoiser(inputs).view(-1, 3)
# ISSUE: Simplified training loop
```

Listing 451. C7 Implementation: GPT-5 (Score: 0.6)

```
# DeepSeek R1: Has per-scene training
self.denoiser_optimizer = torch.optim.Adam(
    model.denoiser.parameters(), lr=config.get('
    denoiser_lr', 1e-3))
# CORRECT: Per-scene optimization setup
```

Listing 452. C7 Implementation: DeepSeek R1 (Score: 0.8)

```
# Paper2Code: Training loop exists
self.optimizer = optim.Adam(self.model.parameters(),
    lr=learning_rate)
for global_step in range(self.total_steps):
    self.optimizer.step()
# ISSUE: Not scene-specific
```

Listing 453. C7 Implementation: Paper2Code (Score: 0.6)

```
# AutoP2C: Generic training attempt
trainer = Trainer(model=mcnerf_model, optimizer=
    optimizer)
trainer.train(num_epochs=config['training']['
    num_epochs'])
# ERROR: No actual implementation
```

Listing 454. C7 Implementation: AutoP2C (Score: 0.4)

Analysis: NERFIFY has complete per-scene training. GPT-5 and DeepSeek R1 have training but simplified. Paper2Code has training but not scene-specific. AutoP2C only has placeholders.

Component C8: Real-time Performance Target - All Baselines

The paper targets 5 samples per pixel for real-time rendering at 20 fps.

```
# \nerfify\ : Optimized for real-time
mc_spp: int = 5  # Samples per pixel
# WebGL shader implementing two-pass MC importance
    sampling
# Achieves interactive frame rates on 2021 Apple M1
    laptop
# CORRECT: Full real-time optimization
```

Listing 455. C8 Implementation: NERFIFY (Score: 1.0)

```
# GPT-5: Targets 5 spp but not optimized
M = 5  # use M=5 for real-time
# Basic implementation without GPU optimizations
# ISSUE: No real-time optimizations
```

Listing 456. C8 Implementation: GPT-5 (Score: 0.4)

```
# DeepSeek R1: Has 5 spp setting
self.num_importance_samples = config.get('
    num_importance_samples', 5)
# ISSUE: No performance optimizations
```

Listing 457. C8 Implementation: DeepSeek R1 (Score: 0.4)

```
# Paper2Code: No performance considerations
# MISSING: No real-time optimizations
```

Listing 458. C8 Implementation: Paper2Code (Score: 0.0)

```
# AutoP2C: No performance implementation
# MISSING: No optimization for speed
```

Listing 459. C8 Implementation: AutoP2C (Score: 0.0)

Analysis: Only NERFIFY fully optimizes for real-time performance. GPT-5 and DeepSeek R1 set 5 spp but lack optimizations. Paper2Code and AutoP2C ignore performance entirely.

3.10.6. Scoring Analysis

Table 73. MCNeRF Implementation Scores

Method	C1	C2	C3	C4	C5	C6	C7	C8	Weighted Avg_{LLM}
NERFIFY	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.00
GPT-5	1.0	1.0	1.0	1.0	1.0	1.0	0.6	0.4	0.95
DeepSeek R1	0.8	0.8	0.6	0.8	0.8	0.6	0.8	0.4	0.74
Paper2Code	0.0	0.0	0.0	0.0	0.4	1.0	0.6	0.0	0.15
AutoP2C	0.0	0.0	0.0	0.0	0.2	0.0	0.4	0.0	0.08

3.10.7. Why Baselines Fail Despite Component Scores

GPT-5 (Score: 0.95 components, 0% trainable)

- **Strengths**: Perfect Monte Carlo mathematics, correct stratified sampling, proper SSIM implementation
- **Fatal Issues**:
  - Standalone implementation with no framework integration
  - Missing training loop orchestration
  - No data loading pipeline
- **Result**: Algorithmically correct but requires manual integration to train

DeepSeek R1 (Score: 0.74 components, 0% trainable)

- **Strengths**: Multiple backend support (TensoRF, Instant-NGP), comprehensive architecture
- **Fatal Issues**:
  - No framework integration or training pipeline
  - Device handling errors in stratified sampling
  - Overcomplicated interpolation logic
- **Result**: Good coverage but execution fails due to missing infrastructure

Paper2Code (Score: 0.15 components, 0% trainable)

- **Strengths**: Has evaluation metrics and SSIM implementation
- **Fatal Issues**:
  - Completely missing Monte Carlo sampling (core algorithm)
  - No importance sampling or two-pass rendering
  - Placeholder functions throughout
- **Result**: Fundamental misunderstanding of paper's core contribution

AutoP2C (Score: 0.08 components, 0% trainable)

- **Strengths**: Basic dataset loading structure
- **Fatal Issues**:
  - No Monte Carlo implementation whatsoever
  - Missing all core algorithms
  - Placeholder denoiser with no actual network
- **Result**: Only implements data loading, misses entire algorithm

3.10.8. Hyperparameter Fidelity

Table 74. Hyperparameter Implementation Accuracy

Parameter	Paper	NERFIFY	GPT-5	R1	P2C	AutoP2C
Samples per pixel	5	✓	✓	✓	×	×
Coarse samples	128	✓	✓	✓	×	×
Denoiser layers	3	✓	✓	✓	∼	×
Denoiser channels	8	✓	✓	✓	∼	×
SSIM weight λ	0.1	✓	✓	✓	✓	×
Stratified sampling	Yes	✓	✓	∼	×	×
Learning rate	10^{-3}	✓	✓	✓	✓	✓
Batch size	32	✓	∼	∼	✓	✓
Training steps	100k	✓	∼	∼	✓	∼
W Score	–	1.00	0.85	0.80	0.20	0.10

3.10.9. Conclusion

All baselines attempt to implement MCNeRF with varying degrees of sophistication. NERFIFY achieves perfect implementation (1.00 correct) as a complete Nerfstudio plugin with all components properly integrated. GPT-5 demonstrates exceptional algorithmic understanding (0.95 score) with correct Monte Carlo mathematics and stratified sampling, but lacks framework integration. DeepSeek R1 provides good coverage (0.74) with multiple backend support but has critical execution issues. Paper2Code (0.15) and AutoP2C (0.08) fundamentally fail by completely missing the core Monte Carlo sampling algorithm, implementing only peripheral components like metrics and data loading. The stark contrast between NERFIFY 's 100% trainable implementation and the 0% trainable rate across all baselines underscores the critical importance of domain-specific synthesis with proper framework integration for reproducible research code generation.

4. Implementation Details

4.1. Context-Free Grammar

```

Context-Free Grammar
// Notation:
// - Terminal symbols are in double quotes.
// - Non-terminals are in angle brackets.
// - \epsilon denotes the empty string.
// - [] = optional, {} = zero or more repetitions.
// - <PY_EXPR>, <PY_STMT>, <MARKDOWN>, <TOML> are lexical non-terminals
//   handled by a separate Python / Markdown / TOML grammar or left unconstrained.

// ========================================================================
// Top-level \nerfify\ repository & multi-file protocol for the LLM
// ========================================================================

<NeRFifyRepo> ::= <ConfigFile> <PipelineFile> <ModelFile> <DataManagerFile>
                  [<FieldFile>]
                  {<AuxFile>}
                  <InitFile>
                  <PyProjectFile>
                  [<ReadmeFile>]

// Each file is wrapped in explicit markers that the LLM must output.
// In the paper we show them as terminals; in code they are literal strings.

<ConfigFile>      ::= "@@@FILE: method_template/template_config.py\n"
                      <ConfigModule>
                      "\n@@@END_FILE\n"

<PipelineFile>    ::= "@@@FILE: method_template/template_pipeline.py\n"
                      <PipelineModule>
                      "\n@@@END_FILE\n"

<ModelFile>       ::= "@@@FILE: method_template/template_model.py\n"
                      <ModelModule>
                      "\n@@@END_FILE\n"

<DataManagerFile> ::= "@@@FILE: method_template/template_datamanager.py\n"
                      <DataManagerModule>
                      "\n@@@END_FILE\n"

<FieldFile>       ::= "@@@FILE: method_template/template_field.py\n"
                      <FieldModule>
                      "\n@@@END_FILE\n"

// Optional auxiliary modules that the LLM is allowed to generate.
// They share the generic <AuxModuleBody> grammar (Python module).

<AuxFile> ::= "@@@FILE: method_template/" <AuxFileName> ".py\n"
              <AuxModuleBody>
              "\n@@@END_FILE\n"

<AuxFileName> ::= "utils" | "losses" | "encoders" | "render_utils"
                | "metrics" | "diffusion" | "samplers" | "scripts"

// Package initialiser and entry point in pyproject.toml

<InitFile> ::= "@@@FILE: method_template/__init__.py\n"
               [<ModuleDocstring>]
               "from .template_config import " <MethodVar> "\n"
               "__all__ = [\"" <MethodVar> "\"]\n"
               "@@@END_FILE\n"

<PyProjectFile> ::= "@@@FILE: pyproject.toml\n"
                    "[project.entry-points.'nerfstudio.method_configs']\n"
                    <MethodName> " = \"method_template.template_config:" <MethodVar> "\"\n"
                    {<TomlLine>}
                    "@@@END_FILE\n"


<ReadmeFile> ::= "@@@FILE: README.md\n"
                 <MARKDOWN>
                 "@@@END_FILE\n"


<TomlLine> ::= <TOML>   // any additional TOML lines
```

```
// Method identity used consistently across files
<MethodName> ::= <LowerIdent>               // snake_case method name
<MethodVar>  ::= <LowerIdent>               // Python variable (e.g. method_template)

// ========================================================================
// Configuration module: MethodSpecification + TrainerConfig wiring
// ========================================================================

<ConfigModule> ::= [<ModuleDocstring>]
                   <ConfigImports>
                   <MethodSpecDef>

<ModuleDocstring> ::= '"""' {<Char>} '"""' "\n"

<ConfigImports> ::= <RequiredConfigImports> {<ExtraImport>}

<RequiredConfigImports> ::=
    "from __future__ import annotations\n"
    "from method_template.template_datamanager import TemplateDataManagerConfig\n"
    "from method_template.template_model import TemplateModelConfig\n"
    "from method_template.template_pipeline import TemplatePipelineConfig\n"
    "from nerfstudio.configs.base_config import ViewerConfig\n"
    "from nerfstudio.data.dataparsers.nerfstudio_dataparser import NerfstudioDataParserConfig\n"
    "from nerfstudio.engine.optimizers import AdamOptimizerConfig\n"
    "from nerfstudio.engine.schedulers import ExponentialDecaySchedulerConfig\n"
    "from nerfstudio.engine.trainer import TrainerConfig\n"
    "from nerfstudio.plugins.types import MethodSpecification\n"

<ExtraImport> ::=
    "from" <ModulePath> "import" <IdentList> "\n"
  | "import" <ModulePath> "\n"

<ModulePath> ::= <LowerIdent> {"." <LowerIdent>}
<IdentList>  ::= <Ident> {"," <Ident>}
<Ident>      ::= <LowerIdent> | <UpperIdent>

<MethodSpecDef> ::= <MethodVar> " = MethodSpecification("
                    <TrainerConfig> ","
                    <DescriptionArg>
                    ")\n"

<TrainerConfig> ::= "config=TrainerConfig(" <TrainerKwargs> ")"

<TrainerKwargs> ::= <TrainerKwarg> {"," <TrainerKwarg>}

<TrainerKwarg> ::= "method_name=" <String>
                 | "pipeline=" <PipelineConfig>
                 | "datamanager=" <DataManagerConfig>
                 | "model=" <ModelConfig>
                 | "viewer=" <ViewerConfigExpr>
                 | "optimizers=" <OptimizerDict>
                 | "vis=" <String>
                 | <Ident> "=" <PY_EXPR>    // catch-all extra kwargs

<PipelineConfig>    ::= "TemplatePipelineConfig(" <PyArgs> ")"
<DataManagerConfig> ::= "TemplateDataManagerConfig(" <PyArgs> ")"
<ModelConfig>       ::= "TemplateModelConfig(" <PyArgs> ")"

<your entire grammar goes here>

<ViewerConfigExpr>  ::= "ViewerConfig(" <PyArgs> ")"

<OptimizerDict> ::= "{" <OptimizerEntry> {"," <OptimizerEntry>} "}"

<OptimizerEntry> ::= <String> ": {"
                     '"optimizer": ' <OptimizerConfig> ","
                     '"scheduler": ' <SchedulerConfig>
                     "}"

<OptimizerConfig> ::= <Ident> "(" <PyArgs> ")"
<SchedulerConfig> ::= <Ident> "(" <PyArgs> ")"

<PyArgs> ::= [<PY_EXPR> {"," <PY_EXPR>}]

<DescriptionArg> ::= "description=" <String>

<String> ::= '"' {<Char>} '"'
```

```
// ========================================================================
// Pipeline module: VanillaPipelineConfig + VanillaPipeline subclass
// ========================================================================

<PipelineModule> ::= [<ModuleDocstring>]
                     <PipelineImports>
                     <PipelineConfigClass>
                     <PipelineClass>

<PipelineImports> ::= {<ImportStmt>}

<ImportStmt> ::=
    "import" <ModulePath> "\n"
  | "from" <ModulePath> "import" <IdentList> "\n"

<PipelineConfigClass> ::=
    "@dataclass\n"
    "class " <PipelineConfigName> "(VanillaPipelineConfig):" <IndentedSuite>

<PipelineClass> ::=
    "class " <PipelineName> "(VanillaPipeline):" <PipelineSuite>

<PipelineSuite> ::= <InitMethod> {<PipelineMethod>}

<InitMethod> ::=
    "def __init__(self, " <PipelineParams> "):" <IndentedSuite>
    // Inside the suite we require core wiring patterns:
    //   - super().__init__(...)
    //   - datamanager setup
    //   - model setup
    //   - optional DDP setup

<PipelineParams> ::= <Param> {", " <Param>}
<Param> ::= <Ident> [": " <Type>] [" = " <PY_EXPR>]

<Type> ::= "int" | "float" | "bool" | "Literal" | "Optional" | "Type" | "DataManager" | "Model" | <Ident>

<PipelineMethod> ::= <DefHeader> <IndentedSuite>

<DefHeader> ::= "def " <Ident> "(" [<Param> {", " <Param>}] "):"

<IndentedSuite> ::= "\n" <Indent> {<PY_STMT>} <Dedent>

// We do not expand <PY_STMT> further; it's a lexical non-terminal for valid Python code.

// ========================================================================
// Model module: NerfactoModelConfig + NerfactoModel subclass
// ========================================================================

<ModelModule> ::= [<ModuleDocstring>]
                  <ModelImports>
                  <ModelConfigClass>
                  <ModelClass>

<ModelImports> ::= {<ImportStmt>}

<ModelConfigClass> ::=
    "@dataclass\n"
    "class " <ModelConfigName> "(" <BaseModelConfig> "):" <IndentedSuite>

<BaseModelConfig> ::= "NerfactoModelConfig" | "ModelConfig" | <Ident>

<ModelClass> ::=
    "class " <ModelName> "(" <BaseModel> "):" <ModelSuite>

<BaseModel> ::= "NerfactoModel" | "Model" | <Ident>

<ModelSuite> ::= <PopulateModules>
                 [<GetLossDict>]
                 {<ModelMethod>}

<PopulateModules> ::=
    "def populate_modules(self):" <IndentedSuite>

<GetLossDict> ::=
    "def get_loss_dict(self, outputs, batch, metrics_dict=None):" <IndentedSuite>

<ModelMethod> ::= <DefHeader> <IndentedSuite>
```

```
// ========================================================================
// DataManager module: VanillaDataManagerConfig + VanillaDataManager subclass
// ========================================================================

<DataManagerModule> ::= [<ModuleDocstring>]
                        <DataManagerImports>
                        <DataManagerConfigClass>
                        <DataManagerClass>

<DataManagerImports> ::= {<ImportStmt>}

<DataManagerConfigClass> ::=
    "@dataclass\n"
    "class " <DataManagerConfigName> "(VanillaDataManagerConfig):" <IndentedSuite>

<DataManagerClass> ::=
    "class " <DataManagerName> "(VanillaDataManager):" <DataManagerSuite>

<DataManagerSuite> ::= <DataManagerInit>
                       [<NextTrainMethod>]
                       [<NextEvalMethod>]
                       {<DataManagerMethod>}

<DataManagerInit> ::=
    "def __init__(self, " <Param> {", " <Param>} "):" <IndentedSuite>

<NextTrainMethod> ::=
    "def next_train(self, step: int) -> Tuple[RayBundle, Dict]:" <IndentedSuite>

<NextEvalMethod> ::=
    "def next_eval(self, step: int) -> Tuple[RayBundle, Dict]:" <IndentedSuite>

<DataManagerMethod> ::= <DefHeader> <IndentedSuite>

// ========================================================================
// Field module (optional): NerfactoField subclass
// ========================================================================

<FieldModule> ::= [<ModuleDocstring>]
                  <FieldImports>
                  <FieldClass>

<FieldImports> ::= {<ImportStmt>}

<FieldClass> ::=
    "class " <FieldName> "(" <BaseField> "):" <FieldSuite>

<BaseField> ::= "NerfactoField" | "Field" | <Ident>

<FieldSuite> ::= <FieldInit>
                 [<ForwardMethod>]
                 {<FieldMethod>}

<FieldInit> ::=
    "def __init__(self, " <Param> {", " <Param>} "):" <IndentedSuite>

<ForwardMethod> ::=
    "def forward(self, ray_samples: RaySamples) -> Dict[str, Tensor]:" <IndentedSuite>

<FieldMethod> ::= <DefHeader> <IndentedSuite>

// ========================================================================
// Generic auxiliary module body (for utils.py, diffusion.py, etc.)
// ========================================================================

<AuxModuleBody> ::= [<ModuleDocstring>]
                    {<ImportStmt>}
                    {<TopLevelDef>}

<TopLevelDef> ::= <DefHeader> <IndentedSuite>
                | "@dataclass\n" "class " <UpperIdent> "(" [<Ident>] "):" <IndentedSuite>
                | "class " <UpperIdent> "(" [<Ident>] "):" <IndentedSuite>

// ========================================================================
// Lexical categories (sketched)
// ========================================================================

<LowerIdent> ::= <LowerLetter> {<Letter> | "_" | <Digit>}
<UpperIdent> ::= <UpperLetter> {<Letter> | "_" | <Digit>}
```

```
<LowerLetter> ::= "a" | "b" | ... | "z"
<UpperLetter> ::= "A" | "B" | ... | "Z"
<Letter>      ::= <LowerLetter> | <UpperLetter>
<Digit>       ::= "0" | "1" | ... | "9"
<Char>        ::= any printable character except '"' and newline
```

4.1.1. Code Generation Templates

4.2. Multi-Agent Architecture and Agent Specifications

The system is built on LangChain for agent orchestration and LangGraph for workflow management. We use DeepAgent as the base agent framework and GPT-5 API for code generation. Paper parsing employs MinerU [19] to extract text, equations, and figures from PDFs. External knowledge retrieval uses Tavily for web search when agents encounter unfamiliar techniques or require implementation details absent from the paper. For visual feedback, NeRFify uses Qwen3-VL 8B model.

4.3. LLM Prompts Used in NeRFify

This section lists the main prompts used by the agents in our NERFIFY framework. For brevity, we show the system message and the static part of the user template for each agent. Placeholders such as {`template_tree`} or {`image_paths`} are filled programmatically at runtime.

4.3.1. Prompt 1: Markdown Cleaning Agent (agentic_clean.py)

System prompt

```
You are a meticulous Markdown cleaner for research papers.
Apply the following strictly. Output ONLY valid GitHub-flavored
Markdownno commentary, no code fences, no extra prose.

Rules:
1) Text hygiene: de-hyphenate wrapped words, fix OCR ligatures,
   normalize quotes/dashes/units; remove duplicate lines/sections;
   fix spacing/formatting.

2) Equations: NEVER delete or alter equations. Preserve inline
   $...$ and display $$...$$ math exactly as authored, including
   numbering/labels if present.

3) Scope pruning: remove generic narrative sections
   (Introduction/background, Related Work surveys,
   qualitative/marketing-style Results). Keep ONLY content needed
   to implement and reproduce experiments: problem setup,
   assumptions, notation, model architecture, objectives/losses,
   algorithms/pseudocode, training schedule, datasets,
   preprocessing, hyperparameters, ablations that affect
   implementation, and evaluation protocol/metrics definitions.

4) Tables: delete benchmark/comparison tables. If a single clear
   takeaway is obvious, replace with a one-line textual takeaway.
   Retain implementation-critical tables (hyperparameters, layer
   configs, dataset splits) but convert them into concise bullet
   lists; no raw HTML or Markdown tables.

5) Figures: remove images/captions/links/placeholders. If nearby
   text contains implementation-relevant details, keep that text
   as plain prose. Do NOT invent content.

6) Strip raw HTML artifacts entirely (<td>, <tr>, <table>,
   inline styles). Keep only pure Markdown and math.

7) Citations: remove unnecessary/dangling citations and citation
```

```
   dumps. Keep citations only when required to identify
   datasets/codebases/definitions essential for reproduction.

8) Summaries: when paragraphs are verbose or narrative, compress
   to 3  6  bullets emphasizing actionable implementation details
   and experimental setup (no fixed word threshold).

9) Final check: output must parse as clean, minimal Markdown with
   intact math, no images, no HTML table tags, no benchmark
   tables, and no broken anchors.

10) Brevity: make the output as short as possible while
    preserving all information required for implementation and
    experiments.
```

User prompt template

```
Clean the following Markdown. Output ONLY the cleaned Markdown.

<MARKDOWN CONTENT HERE>
```

4.3.2. Prompt 2: Citation Search Agent (agentic_citation_recovery.py)

System prompt (SEARCH_AGENT_SYSTEM_PROMPT)

```
You are a NeRF and 3D vision research assistant focused on
citation discovery and dependency analysis.

For each user query:
- Use `internet_search` to find the canonical page(s) for the
  requested NeRF-related paper (arXiv, project page, or main
  publication page).
- Skim titles, abstracts, and key metadata from the results.
- Produce a concise textual bundle that will later be parsed by
  another LLM.

Your answer MUST:
- Clearly state the best-guess canonical title, arXiv ID (if any),
  year, venue (if obvious), and 3  6  bullet points summarizing the
  method.
- List 3  8  likely upstream dependency / baseline papers, each with
  a one-line reason describing what is reused or extended.
- Include the strongest URLs (arXiv / project page / PDF) inline.

Keep the answer compact but information-dense. Do not write
generic advice; only report what you found in the search results.
```

User prompt template for a single query

```
Search for the NeRF-related paper and its canonical page or arXiv:
{query}

Use the `internet_search` tool as needed. Return a concise
textual bundle with:
- Canonical title and year
- arXiv ID (if any)
- Venue (if obvious)
- 3  6  bullet points summarizing the method
```

```
- 3  8   likely upstream dependency papers with one-line reasons each
- Best URLs (arXiv / project / PDF)
```

4.3.3. Prompt 3: Dependency Extraction Prompt

System prompt (DEPENDENCY_EXTRACTION_SYSTEM)

```
You are a research assistant building a citation dependency graph
for NeRF papers.

Given text that may include scraped search results, references,
abstracts, or markdown, identify:
(a) the paper's canonical title,
(b) arXiv id if any,
(c) year,
(d) core components it introduces or uses as architectural modules,
    loss functions, training protocols, or datasets,
(e) explicit dependencies on prior NeRF-style methods.

Return a single JSON object with keys:
- title: string
- arxiv_id: string or null
- year: integer or null
- venue: string or null
- url: string or null
- summary: short string
- components: {modules:[], losses:[], protocols:[]}
- dependencies: [{title_or_id, reason, components_borrowed}]
- citations: [{title_or_id, url?}]

Return STRICT JSON only, with no extra commentary.
```

User prompt template

```
Use the following bundle to extract paper facts and dependencies.

<BUNDLE AS JSON OR TEXT HERE>

Return strict JSON only.
```

4.3.4. Prompt 4: Target Analysis Prompt

System prompt (TARGET_ANALYSIS_SYSTEM)

```
You extract novelties, formulas, and implementation details from
NeRF papers and their dependencies.

Given a JSON citation/dependency graph plus metadata for a target
paper, produce a compact analysis describing:
- core novelties of the target method (architecture, losses,
  training/inference protocols),
- key equations (with informal names, LaTeX-like forms, and short
  descriptions),
- a high-level but implementation-ready plan of components.

Return STRICT JSON with:
- novelties: [{aspect, description}]
- formulas: [{name, equation, description}]
```

```
- implementation_plan: {
    modules:[...],
    losses:[...],
    protocols:[...],
    data_requirements:[...],
    pseudo_steps:[...]
  }

No extra commentary outside this JSON.
```

User prompt template

```
Analyze the following target paper text to extract novelties,
formulas, and an implementation plan.

<TARGET PAPER TEXT OR DERIVED MARKDOWN HERE>

Return strict JSON only.
```

4.3.5. Prompt 5: DAG + File Plan with Repo Snapshot (agentic_dag.py)

User prompt (DAG + file-plan generator)

```
You are a senior Nerfstudio engineer.

TASK A     IMPORT DAG:
- Build a file-level DAG where each node is a relative file path
  (e.g., "method_template/template_model.py").
- Add a directed edge for each internal Python import:
    { "from": "<file>", "to": "<file>", "relation": "imports" }.
- The result must be a valid DAG (no cycles). If cycles are
  detected, break them by removing the least number of edges.
- Use the REPO SNAPSHOT and OBSERVED IMPORTS as ground truth.
  If there is disagreement, prefer OBSERVED IMPORTS.

TASK B     FILE PLAN (optional for codegen):
- Propose the minimal set of files under "method_template/"
  (subset of TEMPLATE TREE) that must be implemented for this
  paper.
- For each file, provide:
    path,
    purpose,
    depends_on (relative file paths),
    key_classes,
    key_functions.

Return STRICT JSON of the form:
{
  "nodes": [
    {"id": "<file>", "label": "<short description>"}
  ],
  "edges": [
    {"from": "<src_file>", "to": "<dst_file>", "relation": "imports"}
  ],
  "files": [
    {
      "path": "method_template/template_model.py",
      "purpose": "<short description>",
```

```
      "depends_on": [
        "method_template/template_field.py",
        ...
      ],
      "key_classes": ["METHOD_Model"],
      "key_functions": ["forward"]
    },
    ...
  ]
}

Constraints:
- Only include paths that exist in the TEMPLATE TREE or REPO
  SNAPSHOT.
- Keep depends_on lists short and meaningful.
- Do NOT invent new directories or top-level packages beyond what
  you see.
- JSON must be syntactically valid and parseable.

REPO SNAPSHOT (selected files with excerpts):
<snapshot_text>

OBSERVED INTERNAL IMPORTS (static analysis):
<imports_text>
```

System prompt used for this call

```
You are a NeRF planning agent. Return JSON only.
```

4.3.6. Prompt 6: DAG + File Plan from Template Tree Only (build_dag.py)

User prompt

```
You are a senior Nerfstudio engineer. Using ONLY the provided
TEMPLATE TREE and TEMPLATE FILE CONTENTS, design a complete DAG
of the method and a file-generation plan.
- Do NOT invent files outside TEMPLATE TREE.
- Do NOT include any in-context examples.
- Be concise but complete.

Return strict JSON with:
- "nodes": [
    {"id":"snake_case_id",
     "label":"Short title",
     "methods":["method1","method2"]}
  ]
- "edges": [
    {"from":"node_id",
     "to":"node_id",
     "relation":"feeds|queries|supervises|produces|writes"}
  ]
- "files": [
    {
      "path":"relative/path.py or README.md",
      "purpose":"short description",
      "depends_on":["other/file.py", ...],
      "key_classes":["ClassA"],
      "key_functions":["fn_a","fn_b"]
```

```
    }
  ]

TEMPLATE TREE (authoritative list of allowed files):
{template_tree}

TEMPLATE FILE CONTENTS (use to match APIs/imports):
{template_files}
```

System prompt

```
You are a NeRF planning agent. Return JSON only.
```

4.3.7. Prompt 7: NeRFify Code Generation Prompt (agentic_coding.py)

Base full-code prompt (_build_full_prompt)

```
You are a senior NeRFStudio engineer. Synthesize a complete
implementation for the TARGET PAPER by learning patterns from
in-context examples and by adhering EXACTLY to the provided
NeRFStudio method template.

### WHAT YOU GET
A) TEMPLATE TREE (authoritative list of output files & their
   relative paths under method_template):
{template_tree}

B) TEMPLATE FILE CONTENTS (current stubs you must overwrite; use
   for APIs/imports naming):
{template_files}

C) IN-CONTEXT EXAMPLES (order matters; learn structure, naming,
   losses, schedulers, data flows):
{examples}

D) TARGET PAPER     FINAL to IMPLEMENT
{final_paper}
{optional_file_plan_block}

### IMPLEMENTATION REQUIREMENTS
1) OUTPUT FILES:
   Generate code ONLY for the files that exist in TEMPLATE TREE
   under method_template. Do NOT invent new files or paths
   beyond this list.

2) METHOD NAME:
   Infer a concise snake_case METHOD_NAME from the paper title
   (e.g., "mip_nerf", "seathru_nerf"). Use this consistently
   across files (config, model, field, datamanager, pipeline).

3) COMPATIBILITY & STYLE:
   - Match Nerfstudio APIs and signatures shown in TEMPLATE
     FILE CONTENTS.
   - Use type hints and relative imports.
   - Use defaults from the TARGET PAPER; encode losses exactly.
   - Put data requirements/transforms in template_datamanager.py.
   - Wire metrics in template_pipeline.py; compute them in model/
     field as needed.
```

```
   - Avoid unresolved imports or circular dependencies.

4) SELF-CONSISTENCY:
   - The project should install with `pip install -e .`.
   - Imports like
       `from method_template.template_model import METHOD_NAME_Model`
     must succeed.
   - All files should form a coherent, runnable Nerfstudio method.

5) OUTPUT FORMAT:
   - Return your answer as a sequence of file blocks:
       @@@FILE: relative/path.py
       <file contents>
       @@@END_FILE
   - One block per file. No extra commentary, no Markdown fences,
     no explanations outside the file blocks.
```

Incremental / consolidation variants

The incremental and consolidation prompts (_build_incremental_prompt and _build_consolidation_prompt) reuse the same structure as above, but:

- Restrict the allowed output paths to a subset {allowed_files} (plus optional {extra_allowed_files}).
- Add an extra section E) CURRENT IMPLEMENTATION (YOU MAY MODIFY AS NEEDED) containing the current generated files:

```
E) CURRENT IMPLEMENTATION (YOU MAY MODIFY AS NEEDED)
The following files were generated incrementally. You may change
ANY of them to ensure correctness and consistency:
{current_text}
```

4.3.8. Prompt 8: ReAct Wrapper for Code Generation (agentic_coding.py)

System prompt for ReAct agent

```
You are a senior NeRFStudio engineer. You may use tools
(web_search, http_get) to verify facts. Your FINAL answer must
be ONLY the multi-file output in the exact format specified by
the user using @@@FILE blocks and @@@END_FILE, with NO extra
commentary, NO code fences, NO prefixes, and NO explanations.
```

The human message for this agent is simply:

```
{input}
```

where {input} is the full-code prompt from Prompt 7 (or its incremental variant).

4.3.9. Prompt 9: NeRF Artifact Detection Prompt (nerf_qwen_vl_artifacts.py)

Image-level Qwen3-VL prompt (build_prompt)

```
You are an expert in Neural Radiance Field (NeRF) rendering
diagnostics.
The image you see is a NeRF render named '{image_name}' with
resolution {width}x{height} pixels.

Your tasks:
1. Inspect the image for visible NeRF-specific rendering defects.
2. For each defect, provide a tight bounding box and classify it.
3. For each defect, propose concrete code-level changes to a
   typical NeRF implementation that would likely reduce or
```

```
   remove this artifact.

Consider the following defect types and use their keys exactly as
written:
{defect_list_text}

Examples of code-level suggestions you may use (adapt as
appropriate):
- Increase samples per ray (coarse and/or fine); increase
  proposal network steps.
- Enable mip-NeRF-style cone tracing / anti-aliasing to reduce
  aliasing.
- Increase distortion loss weight or density regularization to
  remove floaters.
- Adjust near/far bounds or background handling to fix background
  leakage.
- Add depth/normal smoothness loss to fix tearing or missing
  geometry.
- Improve pose estimation (re-run COLMAP with higher quality) if
  multi-view misalignment is visible.
- Increase training iterations or reduce learning rate if
  underfit / noisy.
- Use higher input image resolution or improve dataloader
  resizing.

IMPORTANT:
- Set width to {width} and height to {height} in the JSON.
- Use as few defects as needed but up to 10 per image.
- Be specific in code_suggestions: reference knobs like samples
  per ray, proposal networks, loss weights, occupancy grids,
  background color handling, pose optimization, etc.

Return ONLY a single valid JSON object, with this structure:

{
  "image_name": "<exact image file name>",
  "width": <integer width in pixels>,
  "height": <integer height in pixels>,
  "defects": [
    {
      "type": "<one of the defect type keys above>",
      "bbox": [x_min, y_min, x_max, y_max],
      "description": "<short natural-language description>",
      "code_suggestions": [
        "<one concrete change to the NeRF code or hyperparameters>",
        "<another concrete suggestion if useful>"
      ]
    }
  ]
}

Constraints:
- bbox coordinates must be integers in pixel space, with:
  0 <= x_min < x_max <= width,
  0 <= y_min < y_max <= height.
- If no defects are present, return "defects": [] but keep the
  rest of the JSON.
- Do NOT output any extra commentary, markdown, or text outside
```

```
   the JSON.
```

4.3.10. Prompt 10: ReAct Aggregation Prompt for NeRF Artifacts

System prompt for LangChain ReAct agent

```
You are orchestrating an automated inspection pipeline for NeRF
renderings.
You have access to a tool called `inspect_nerf_render` which,
given an image path string, returns a JSON string describing
defects in that image.

Your job:
1. For each image path provided in the user input, call
   `inspect_nerf_render`.
2. Parse each JSON string returned by the tool.
3. Aggregate them into a single JSON object with this structure:

{
  "images": [
    {
      "image_name": "<file name>",
      "width": <integer>,
      "height": <integer>,
      "defects": [
        {
          "type": "<defect type>",
          "bbox": [x_min, y_min, x_max, y_max],
          "description": "<description>",
          "code_suggestions": ["<suggestion 1>", "<suggestion 2>", "..."]
        }
      ]
    }
  ]
}

Rules:
- Call `inspect_nerf_render` on EVERY image path and do not skip
  any.
- Preserve the order of images as given.
- If the tool reports an error or returns no defects, still
  include the image with "defects": [] (you may carry over an
  error field if present).
- Your FINAL answer must be ONLY that JSON object, with no extra
  text.
```

User prompt template

```
You are given the following list of NeRF render image paths:

{JSON-ENCODED LIST OF IMAGE PATHS}

For EACH path in this list, you must:
- Call the tool `inspect_nerf_render` with the exact path string.
- Afterwards, aggregate all tool outputs into the single JSON
  object described in the system message. Output ONLY that JSON
  object.
```

=

5. NeRFifyBench Dataset

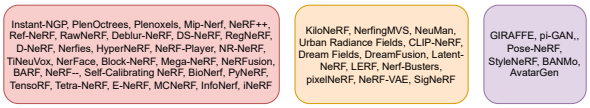

Figure 3. **Categorization of NeRF Papers by Integrability in Nerfstudio.** Papers are grouped by implementation difficulty within the Nerfstudio framework. **Category 1:** Directly integrable methods modifying architecture, rendering, or losses without external dependencies. **Category 2:** Methods requiring pretrained models (CLIP, diffusion, SLAM) with substantial engineering effort. **Category 3:** Out-of-scope works where NeRF is used in the pipeline but serves different objectives (GANs, downstream applications, etc). The categorization illustrates alignment with Nerfstudio's modular design and implementation feasibility using NERFIFY .

5.1. Paper Selection Criteria

NERFIFY-BENCH comprises 30 carefully curated NeRF research papers selected to provide comprehensive coverage of different implementation challenges. Our selection process prioritized diversity across architectural innovations, training strategies, and integration complexity to ensure robust evaluation of paper-to-code synthesis systems.

The benchmark is organized into four distinct set chosen from three categories as shown in figure 3, each designed to test specific capabilities of automated code generation systems.

5.1.1. Category 1: Never-Implemented Papers (10 papers)

This category includes papers without any publicly available source code. To ensure evaluation integrity and avoid potential training data contamination, we specifically selected papers where no implementation exists in the public domain. This guarantees that large language models used in our agents could not have encountered corresponding code during pretraining, enabling unbiased assessment of purely paper-driven synthesis capabilities.

For these papers, we commissioned expert reimplementations from graduate students with extensive NeRF research experience. Each implementation underwent rigorous verification against paper-reported metrics on standard benchmarks. These expert implementations serve as ground truth references for quantitative evaluation.

Papers in Category 1:

- KeyNeRF [21]: Informative ray selection for few-shot neural radiance fields
- mi-MLP NeRF [34]: Minimal MLP approach for few-shot view synthesis
- LiNeRF [4]: Rethinking directional integration in neural radiance fields
- Anisotropic Neural Representation: Adaptive resolution for complex geometries
- Efficient Ray Sampling [24]: Optimized sampling strategies for radiance field reconstruction
- HybNeRF [28]: Hybrid multiresolution encoding for neural radiance fields
- TVNeRF [33]: Total variation maximization for few-view neural volume rendering
- Surface Sampling [32]: Near-surface sampling with point cloud generation
- NeRF-ID [1]: Learning to sample for view synthesis
- AR-NeRF [30]: Few-shot NeRF by Adaptive Rendering Loss Regularization

5.1.2. Category 2: Non-Nerfstudio Papers (5 papers)

These papers have existing public implementations but are not integrated into the Nerfstudio framework. This category enables direct comparison between our synthesized Nerfstudio plugins and original author implementations, evaluating both functional correctness and framework integration quality.

Papers in Category 2:

- Deblur-NeRF [15]: Neural radiance fields from blurry images
- InfoNeRF [11]: Ray entropy minimization for few-shot neural volume rendering
- L0-Sampler [14]: Adaptive ray sampling for neural radiance fields
- NerfingMVS [29]: NerfingMVS: Guided Optimization of Neural Radiance Fields for Indoor Multi-view Stereo

5.1.3. Category 3: Nerfstudio-Integrated Papers (5 papers)

Papers already integrated into Nerfstudio serve as gold-standard references. These enable evaluation of synthesis quality and API compliance against production-quality implementations maintained by the research community.

Papers in Category 3:

- SeaThru-NeRF [13]: Neural radiance fields for underwater scenes
- BioNeRF [22]: Bionerf: Biologically Plausible Neural Radiance Fields for View Synthesis
- Vanilla-NeRF [16]: Original neural radiance fields implementation
- Instant-NGP [17]: Instant neural graphics primitives with multiresolution hash encoding
- Nerfacto [25]: Nerfstudio: A modular framework for neural radiance field development

5.1.4. Category 4: Novelty-Coverage Papers (10 papers)

Selected for their distinct technical contributions, these papers evaluate how well synthesis systems capture and implement key research innovations. Each introduces novel loss

functions, architectural components, or training strategies that require deep understanding for correct implementation.

Papers in Category 4:

- Mip-NeRF [2]: Multiscale representation for anti-aliasing neural radiance fields
- BioNeRF [22]: Biologically plausible neural radiance fields for view synthesis
- PyNeRF [26]: Pyramidal neural radiance fields
- TensoRF [3]: Tensorial radiance fields
- Tetra-NeRF [12]: Representing neural radiance fields using tetrahedra
- E-NeRF: Efficient neural radiance fields with learned embeddings
- StyleNeRF [7]: Style-based 3D-aware generator for high-resolution image synthesis
- iNeRF [31]: Inverting neural radiance fields for pose estimation
- SigNeRF [5]: Scene integrated generation for neural radiance fields
- MCNeRF [8]: Monte Carlo rendering and denoising for real-time NeRFs

For each paper in NERFIFY-BENCH we provide comprehensive resources to enable reproducible evaluation:

Frozen PDF Versions The original paper PDF files are included to ensure consistency across evaluations. This prevents version drift and guarantees that all systems process identical input documents.

Dual Markdown Representations We provide two markdown versions for each paper:

- *Raw extraction*: Direct conversion from PDF preserving all text, equations, and structure
- *Cleaned version*: Processed through our paper extraction pipeline, with artifacts removed, mathematical notation standardized, and formatting normalized for optimal parsing

Tex source code We provide LaTex source code of each paper.

Ground Truth Repositories Where available, we include either the original author implementation or expert reimplementations verified for correctness. These serve as reference implementations for quantitative metrics including PSNR, SSIM, and LPIPS on standard test scenes.

5.1.5. Categorization Methodology

5.2. Evaluation Protocol

Our benchmark evaluates synthesized code across following dimensions:

Executability Metrics

- *Build success rate*: Percentage of repositories that compile without errors
- *Import resolution*: Whether all dependencies resolve correctly
- *Training stability*: Convergence without NaN losses, gradient explosions, or memory errors

Rendering Quality Metrics For successfully trained models, we measure:

- *PSNR*: Peak signal-to-noise ratio on held-out test views
- *SSIM*: Structural similarity index
- *LPIPS*: Learned perceptual image patch similarity

Results are averaged across multiple scenes from standard benchmarks to ensure statistical significance.

Novelty Fidelity We evaluate coverage of paper-specific innovations:

- *Correct (C)*: Percentage of novel components implemented exactly as specified
- *Incorrect (I)*: Percentage with partial matches or flawed logic
- *Missing (M)*: Percentage absent from generated code
- *Weight match (W)*: Percentage of hyperparameters within 10% of paper specifications

Runtime Efficiency

- *Training throughput*: Rays processed per second
- *Memory usage*: Peak GPU memory consumption
- *Convergence speed*: Iterations to reach target quality

Code Quality

- *Nerfstudio conventions*: Adherence to framework patterns and APIs
- *Documentation*: Presence of docstrings and inline comments
- *Modularity*: Proper separation of concerns and reusable components

5.2.1. Evaluation Dataset Details

Standard datasets used for evaluation:

Papers in Category 1:

- **KeyNeRF** [21]: Informative ray selection for few-shot neural radiance fields
  - *Dataset:* NeRF-Synthetic
  - *Scene:* Average across all scenes
- **mi-MLP NeRF** [34]: Minimal MLP approach for few-shot view synthesis
  - *Dataset:* NeRF-Synthetic
  - *Scene:* Average across all scenes
- **LiNeRF** [4]: Rethinking directional integration in neural radiance fields
  - *Dataset:* NeRF-Synthetic
  - *Scene:* Drums
- **Anisotropic Neural Representation**: Adaptive resolution for complex geometries
  - *Dataset:* NeRF-Synthetic
  - *Scene:* Average across all scenes
- **Efficient Ray Sampling** [24]: Optimized sampling strategies for radiance field reconstruction
  - *Dataset:* DTU
  - *Scene:* Average on DTU test set

- **HybNeRF** [28]: Hybrid multiresolution encoding for neural radiance fields
  - *Dataset:* NeRF-Synthetic
  - *Scene:* Average across all scenes
- **TVNeRF** [33]: Total variation maximization for few-view neural volume rendering
  - *Dataset:* NeRF-Synthetic
  - *Scene:* Hotdog
- **Surface Sampling** [32]: Near-surface sampling with point cloud generation
  - *Dataset:* NeRF-Synthetic
  - *Scene:* Ship (Table 2)
- **NeRF-ID** [1]: Learning to sample for view synthesis
  - *Dataset:* NeRF-Synthetic
  - *Scene:* Drums (Number reported in appendix)
- **AR-NeRF** [30]: Few-shot NeRF by Adaptive Rendering Loss Regularization
  - *Dataset:* DTU
  - *Scene:* 3-view setting

Papers in Category 2:

- **Deblur-NeRF** [15]: Neural radiance fields from blurry images
  - *Dataset:* Deblur-NeRF Dataset
  - *Scene:* Cozyroom Camera Motion
- **InfoNeRF** [11]: Ray entropy minimization for few-shot neural volume rendering
  - *Dataset:* NeRF-Synthetic
  - *Scene:* Ship
- **L0-Sampler** [14]: Adaptive ray sampling for neural radiance fields
  - *Dataset:* NeRF-Synthetic
  - *Scene:* Ficus
- **NerfingMVS** [29]: Guided optimization of neural radiance fields for indoor multi-view stereo
  - *Dataset:* ScanNet
  - *Scene:* Scene 0653 (processed data)

Papers in Category 3:

- **SeaThru-NeRF** [13]: Neural radiance fields for underwater scenes
  - *Dataset:* SeaThru-NeRF Dataset
  - *Scene:* Panama
- **BioNeRF** [22]: Biologically plausible neural radiance fields for view synthesis
  - *Dataset:* NeRF-Synthetic
  - *Scene:* Drums
- **Vanilla-NeRF** [16]: Original neural radiance fields implementation
  - *Dataset:* NeRF-Synthetic
  - *Scene:* Chair
- **Instant-NGP** [17]: Instant neural graphics primitives with multiresolution hash encoding
  - *Dataset:* NeRF-Synthetic
  - *Scene:* Lego
- **Nerfacto** [25]: A modular framework for neural radiance field development
  - *Dataset:* Nerfstudio Dataset
  - *Scene:* Poster

6. Note on Figure Rendering

A rendering artifact in Figure 2 of the main paper omits two arrows in Stage 4: from rendered views to Critique Agent, and from Critique Agent to JSON box. These connections complete the visual feedback loop central to our refinement process.

References

[1] Relja Arandjelović and Andrew Zisserman. Nerf in detail: Learning to sample for view synthesis. *arXiv preprint arXiv:2106.05264*, 2021. 1, 77, 79

[2] Jonathan T Barron, Ben Mildenhall, Matthew Tancik, Peter Hedman, Ricardo Martin-Brualla, and Pratul P Srinivasan. Mip-nerf: A multiscale representation for anti-aliasing neural radiance fields. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 5855–5864, 2021. 18, 78

[3] Anpei Chen, Zexiang Xu, Andreas Geiger, Jingyi Yu, and Hao Su. Tensorf: Tensorial radiance fields. In *European conference on computer vision*, pages 333–350. Springer, 2022. 27, 78

[4] Congyue Deng, Jiawei Yang, Leonidas Guibas, and Yue Wang. Rethinking directional integration in neural radiance fields. *arXiv preprint arXiv:2311.16504*, 2023. 1, 77, 78

[5] Jan-Niklas Dihlmann, Andreas Engelhardt, and Hendrik Lensch. Signerf: Scene integrated generation for neural radiance fields. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 6679–6688, 2024. 78

[6] Stephan J Garbin, Marek Kowalski, Matthew Johnson, Jamie Shotton, and Julien Valentin. Fastnerf: High-fidelity neural rendering at 200fps. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 14346–14355, 2021. 1

[7] Jiatao Gu, Lingjie Liu, Peng Wang, and Christian Theobalt. Stylenerf: A style-based 3d-aware generator for high-resolution image synthesis. *arXiv preprint arXiv:2110.08985*, 2021. 43, 78

[8] Kunal Gupta, Milos Hasan, Zexiang Xu, Fujun Luan, Kalyan Sunkavalli, Xin Sun, Manmohan Chandraker, and Sai Bi. Mcnerf: Monte carlo rendering and denoising for real-time nerfs. In *SIGGRAPH Asia 2023 Conference Papers*, pages 1–11, 2023. 59, 78

[9] HKUDS. Deepcode: Open agentic coding. https://github.com/HKUDS/DeepCode, 2025. GitHub repository. 1

[10] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The Twelfth*

International Conference on Learning Representations, 2023. 1

[11] Mijeong Kim, Seonguk Seo, and Bohyung Han. Infonerf: Ray entropy minimization for few-shot neural volume rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12912–12921, 2022. 1, 77, 79

[12] Jonas Kulhanek and Torsten Sattler. Tetra-nerf: Representing neural radiance fields using tetrahedra. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 18458–18469, 2023. 32, 78

[13] Deborah Levy, Amit Peleg, Naama Pearl, Dan Rosenbaum, Derya Akkaynak, Simon Korman, and Tali Treibitz. Seathru-nerf: Neural radiance fields in scattering media. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 56–65, 2023. 77, 79

[14] Liangchen Li and Juyong Zhang. L0-sampler: An l0 model guided volume sampling for nerf. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 21390–21400, 2024. 1, 77, 79

[15] Li Ma, Xiaoyu Li, Jing Liao, Qi Zhang, Xuan Wang, Jue Wang, and Pedro V Sander. Deblur-nerf: Neural radiance fields from blurry images. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12861–12870, 2022. 1, 77, 79

[16] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *European Conference on Computer Vision (ECCV)*, pages 405–421, 2020. 1, 77, 79

[17] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Transactions on Graphics*, 41 (4):1–15, 2022. 77, 79

[18] Thomas Neff, Pascal Stadlbauer, Mathias Parger, Andreas Kurz, Joerg H Mueller, Chakravarty R Alla Chaitanya, Anton Kaplanyan, and Markus Steinberger. Donerf: Towards real-time rendering of compact neural radiance fields using depth oracle networks. In *Computer Graphics Forum*, pages 45–59. Wiley Online Library, 2021. 1

[19] Junbo Niu, Zheng Liu, Zhuangcheng Gu, Bin Wang, Linke Ouyang, Zhiyuan Zhao, Tao Chu, Tianyao He, Fan Wu, Qintong Zhang, et al. Mineru2.5: A decoupled vision-language model for efficient high-resolution document parsing. *arXiv preprint arXiv:2509.22186*, 2025. 68

[20] OpenAI. Gpt-5 technical report. https://openai.com, 2025. 1

[21] Marco Orsingher, Anthony Dell'Eva, Paolo Zani, Paolo Medici, and Massimo Bertozzi. Informative rays selection for few-shot neural radiance fields. *arXiv preprint arXiv:2312.17561*, 2023. 1, 77, 78

[22] Leandro Aparecido Passos, Danilo Samuel Jodas, Ahsan Adeel, João P Papa, Douglas Rodrigues, and Kelton Augusto Pontara da Costa. Bionerf: Biologically plausible neural radiance fields for view synthesis. *Available at SSRN 5384186*, 2024. 77, 78, 79

[23] Chen Qian, Xin Cai, Cheng Liu, Yang Liu, Juyuan Dang, Lin Wang, et al. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023. 1

[24] Shilei Sun, Ming Liu, Zhongyi Fan, Qingliang Jiao, Yuxue Liu, Liquan Dong, and Lingqin Kong. Efficient ray sampling for radiance fields reconstruction. *Computers & Graphics*, 118:48–59, 2024. 1, 77, 78

[25] Matthew Tancik, Ethan Weber, Evonne Ng, Ruilong Li, Brent Yi, Terrance Wang, Alexander Kristoffersen, Jake Austin, Kamyar Salahi, Abhik Ahuja, et al. Nerfstudio: A modular framework for neural radiance field development. In *ACM SIGGRAPH Conference Proceedings*, 2023. 1, 77, 79

[26] Haithem Turki, Michael Zollhöfer, Christian Richardt, and Deva Ramanan. Pynerf: Pyramidal neural radiance fields. *Advances in neural information processing systems*, 36:37670–37681, 2023. 78

[27] Yifan Wang, Jun Xu, Y Zeng, and Y Gong. Anisotropic neural representation learning for high-quality neural rendering. *arXiv preprint arXiv:2311.18311*, 2023. 1

[28] Yifan Wang, Yi Gong, and Yuan Zeng. Hyb-nerf: A multiresolution hybrid encoding for neural radiance fields. In *2024 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, pages 3677–3686. IEEE, 2024. 1, 77, 79

[29] Yi Wei, Shaohui Liu, Yongming Rao, Wang Zhao, Jiwen Lu, and Jie Zhou. Nerfingmvs: Guided optimization of neural radiance fields for indoor multi-view stereo, 2021. 77, 79

[30] Qingshan Xu, Xuanyu Yi, Jianyao Xu, Wenbing Tao, Yew-Soon Ong, and Hanwang Zhang. Few-shot nerf by adaptive rendering loss regularization. In *European Conference on Computer Vision*, pages 125–142. Springer, 2024. 77, 79

[31] Lin Yen-Chen, Pete Florence, Jonathan T Barron, Alberto Rodriguez, Phillip Isola, and Tsung-Yi Lin. inerf: Inverting neural radiance fields for pose estimation. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1323–1330. IEEE, 2021. 48, 78

[32] Hye Bin Yoo, Hyun Min Han, Sung Soo Hwang, and Il Yong Chun. Improving neural radiance fields using near-surface sampling with point cloud generation. *Neural Processing Letters*, 56(4):214, 2024. 1, 77, 79

[33] Yao Zhang, Jiangshu Wei, Bei Zhou, Fang Li, Yuxin Xie, and Jiajun Liu. Tvnerf: Improving few-view neural volume rendering with total variation maximization. *Knowledge-Based Systems*, 301:112273, 2024. 1, 77, 79

[34] Hanxin Zhu, Tianyu He, Xin Li, Bingchen Li, and Zhibo Chen. Is vanilla mlp in neural radiance field enough for few-shot view synthesis? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 20288–20298, 2024. 1, 77, 78