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

Anonymous CVPR submission

Paper ID 17728

Abstract

The proliferation of neural radiance field (NeRF) research requires significant efforts to reimplement papers before building upon them. We introduce NERFIFY*, a multi-agent framework that reliably converts NeRF research papers into trainable Nerfstudio plugins, in contrast to generic paper-to-code methods and frontier models like GPT-5 that usually fail to produce runnable code.* NERFIFY *achieves domain-specific executability through six key innovations: (1) Context-free grammar (CFG): LLM synthesis is constrained by Nerfstudio formalized as a CFG, ensuring generated code satisfies architectural invariants. (2) Graph-of-Thought code synthesis: Specialized multi-file-agents generate repositories in topological dependency order, validating contracts and errors at each node. (3) Compositional citation recovery: Agents automatically retrieve and integrate components (samplers, encoders, proposal networks) from citation graphs of references. (4) Visual feedback: Artifacts are diagnosed through PSNR-minima ROI analysis, cross-view geometric validation, and VLM-guided patching to iteratively improve quality. (5) Knowledge enhancement: Beyond reproduction, methods can be improved with novel optimizations. (6) Benchmarking: An evaluation framework is designed for NeRF paper-to-code synthesis across 30 diverse papers. On papers without public implementations,* NERFIFY *achieves visual quality matching expert human code (±0.5 dB PSNR, ±0.2 SSIM) while reducing implementation time from weeks to minutes.* NERFIFY *demonstrates that a domain-aware design enables code translation for complex vision papers, potentiating accelerated and democratized reproducible research. Code, data and implementations will be publicly released.*

1. Introduction

Since its publication in 2020, the Neural Radiance Fields (NeRF) paper [32] has spawned over 1,000 follow-ups. Yet, unavailability of code or standardized implementations for most of them [50] means each subsequent work requires a significant effort to reimplement existing NeRF papers.

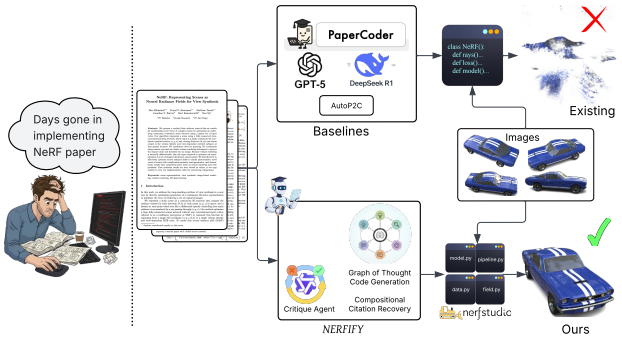

Figure 1. **Overview.** Manual NeRF implementation requires weeks of specialized effort (left). Existing paper-to-code systems fail to produce trainable code. NERFIFY automates this process through grammar-constrained synthesis and compositional citation recovery, generating fully trainable Nerfstudio plugins in minutes (right).

While the issue exists in broader machine learning research [41], effective NeRF implementations present unique challenges. We consider the question of devising a large language model (LLM) agent to automatically produce trainable, performant and standardized NeRFs from their papers.

NeRF implementations are uniquely challenging, with expertise required across volumetric rendering, computer vision and neural optimization. A single misplaced activation or incorrect ray-sphere intersection produces failures ranging from the catastrophic (NaN gradients) to the subtle (degenerate solutions). Debugging requires understanding whether failures stem from code bugs, scene geometry, hyperparameters, or the paper's own ambiguities, with computationally demanding cycles (the original NeRF requires 100k-300k iterations taking 24-48 hours on high-end GPUs [32]).

Generic paper-to-code approaches do not suffice, given that the current best-performing system, O1, achieves only 26.6% accuracy on complex papers compared to 41.4% for human experts [44]. The challenge is exacerbated for NeRFs, where the strong domain coupling needed for its unique combination of rendering mathematics and neural architectures leads to execution failures and large performance gaps. As discussed in Section 2 and exhaustively shown in the Supplementary, recent advances like Paper2Code [42] and AutoP2C

[25] – despite progress on general machine learning tasks – usually do not produce executable code or trainable NeRFs, with issues ranging from incorrect implementations (such as K-Planes with regular MLPs instead of planar factorization), or catastrophic failures (producing only dataset loading code). Another key challenge is that modern NeRF papers build upon chains of dependencies that generic systems cannot resolve, for example, see Figure 3 for K-Planes [10]. A single phrase such as "we adopt the distortion loss from [3]" requires navigating to that paper, locating the correct equation, translating to executable code and implementing stop-gradient operations critical for stable training.

We introduce NERFIFY, a multi-agent framework that reliably converts NeRF research papers into code that trains, converges and matches the visual quality of expert implementations (Figure 1). Its success is based on five key technical innovations. 1) Use of Nerfstudio [47] architecture as a context-free grammar (CFG) that encodes domain-aware module compositions and interface contracts, leading to LLM synthesis of architecturally correct code by construction. 2) Graph-of-thought code generation to coordinate specialized agents that generate multi-file repositories in topological order, validating type signatures, tensor shapes and circular dependencies at each dependency before proceeding. 3) A compositional citation recovery that traverses reference graphs to retrieve implicit dependencies (such as proposal networks from Mip-NeRF 360 [3], hash encoders from Instant-NGP [33]). 4) A critique agent that provides feedback by analyzing training runs to diagnose visual artifacts through PSNR-minima analysis and cross-view geometric validation, using vision-language models (VLMs) to guide targeted fixes. 5) Besides reproduction, an option to improve with optimizations where applicable.

We extensively evaluate with NERFIFY-BENCH, a novel benchmark with 30 diverse papers stratified across code status and implementation complexity. NERFIFY achieves full executability and rendering quality within 0.5 dB PSNR of expert implementations, while generic baselines fail to produce trainable code in 95% of cases.

NERFIFY will accelerate reproducible research, allowing researchers to reproduce complex NeRF papers in hours rather than weeks, rapidly prototype techniques and make accessible to the community papers that would otherwise remain purely theoretical. As the NeRF community continues its growth, NERFIFY ensures that every paper's contributions become available to all researchers, democratizing access to cutting-edge techniques and enabling the compositional research that drives the field forward. With our demonstration that depth and specialization in a multi-agent framework unlocks transformative capabilities for NeRFs, we believe future work will achieve similar transformations for other communities too. Code and data for NERFIFY, NERFIFY-BENCH and generated implementations will be publicly released.

2. Related Work

Generic Paper-to-Code Systems. Paper2Code [42] employs a three-stage pipeline (planning, analysis, generation), but its generic design lacks neural field architectures for NeRF implementation. AutoP2C [25] advances multimodal understanding with high executability on recent papers, but generates non trainable repositories with just placeholders for NeRFs. AutoReproduce [64] introduces paper lineage-extracting domain knowledge from citations. On PaperBench [37], Claude 3.5 Sonnet achieves only 21% accuracy versus 41.4% for human researchers on ICML papers, motivating our domain-specific approach. RPG [26] employs test-driven development with repository planning graphs but cannot effectively parse papers or extract mathematical formulations. Paper2Agent [31] converts papers into conversational interfaces rather than trainable implementations. Recent works [11, 65] explore fine-grained verification and dynamic planning, but remain domain-agnostic.

Multi-Agent Code Generation has emerged as a key field for complex code synthesis [17, 18, 40, 56], with recent systems proposing repository-level code generation [7, 16, 51, 52, 57, 61]. However, general software systems lack the mathematical understanding and architectural constraints required for successful synthesis of research code.

Domain-Specific Code Synthesis outperforms generic approaches by encoding specialized knowledge. Scene Language [63] uses CFG to structure visual program synthesis, CODEP [49] employs Pushdown Automaton and TSL+LLM [30] combines Temporal Stream Logic to act as CFG.

Feedback and Planning Approaches like Graph of Thoughts (GoT) [4] generalize reasoning into directed graphs enabling aggregation, refinement loops, and backtracking, while Tree of Thoughts [58] and Tree-of-Code [35] explore over reasoning trees. Self-Debugging [6], Reflexion [43], Self-Refine [29], Clover [45] and xKG [27] introduce refinement and consistency checks, but either require reference databases or lack domain-specific constraints.

3. Method

3.1. Problem Statement

Given a NeRF research paper $\mathcal{P}$, our objective is to synthesize an executable repository $\mathcal{C}$ that faithfully implements the described method within the Nerfstudio framework. We formalize this paper-to-code synthesis problem as follows.
Repository Definition. A repository $\mathcal{C}$ consists of a set of files and their dependency structure:

$$\mathcal{C} = (F, G), \quad F = \{f_1, f_2, \ldots, f_n\} \tag{1}$$

where $G = \text{BuildRepoDAG}(F)$ is a directed acyclic graph with vertices $V(G) = F$ representing files and edges

CVPR
#17728

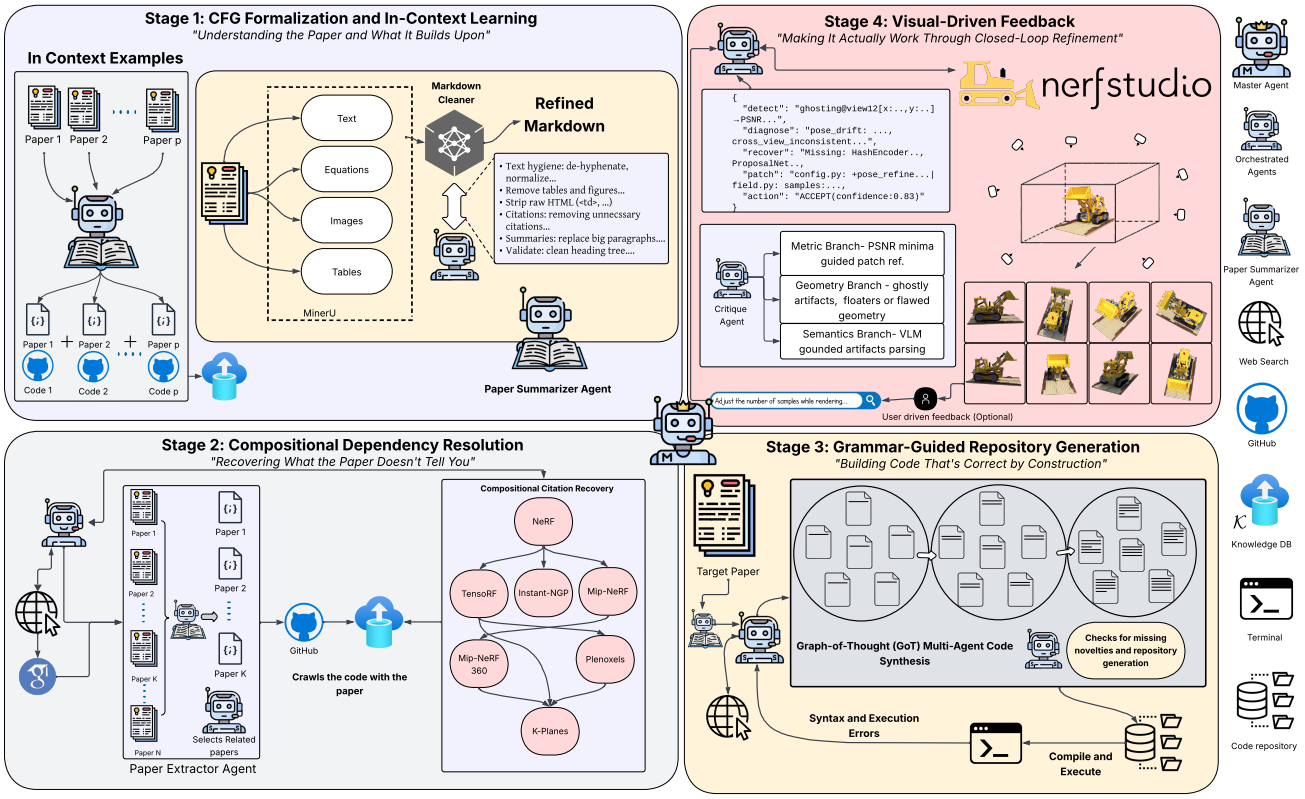

Figure 2. NERFIFY converts NeRF papers into code through four stages: (1) Agent parses and summarizes PDFs into simple markdown, CFG from Nerfstudio and curated paper-code pairs as In-Context examples are saved in $\mathcal{K}$ (2) Compositional dependency resolution traverses citation graphs to retrieve missing components from referenced papers, (3) GoT code synthesis generates repository files through specialized agents operating in topological order (4) Visual refinement iteratively patches artifacts until achieving expert-level quality.

$E(G) \subseteq F \times F$ encoding import and dataflow dependencies. The acyclic constraint ensures compilability: $(f_i, f_j) \in E(G) \implies$ no path exists from f_j to f_i.

Paper Representation We extract structured information from paper $\mathcal{P}$ through a comprehensive parsing pipeline:

$$\mathcal{E}(\mathcal{P}) = \langle T(\mathcal{P}), I(\mathcal{P}), Q(\mathcal{P}), B(\mathcal{P}) \rangle, \quad (2)$$

$$T(\mathcal{P}) = \langle H, \{p_i\}_{i=1}^{n_p}, \{a_\ell\}_{\ell=1}^{n_a}, \{c_k\}_{k=1}^{n_c}, \{r_m\}_{m=1}^{n_r} \rangle. \quad (3)$$

The textual component $T(\mathcal{P})$ encompasses section headings H, paragraphs $\{p_i\}$, algorithm blocks $\{a_\ell\}$, figure captions $\{c_k\}$, and references $\{r_m\}$. Visual content $I(\mathcal{P})$ includes architectural diagrams, figures, and result visualizations that often contain crucial implementation details not present in the text. The mathematical component $Q(\mathcal{P})$ captures equations and formulas that define the core algorithms, while bibliographic metadata $B(\mathcal{P})$ provides citation information essential for compositional recovery.

Agent Formulation Let $\mathcal{A}$ be a multi-agent system that maps paper representations to executable code using auxiliary resources: $\mathcal{A} : (\mathcal{E}(\mathcal{P}); \mathcal{R}) \mapsto \mathcal{C}$. The resource tuple $\mathcal{R} = (\mathcal{K}, \mathcal{W}, \mathcal{X})$ comprises three essential components. The domain knowledge base $\mathcal{K}$ encodes NeRF-specific architectural patterns, common implementation strategies, and framework conventions (CFG) accumulated from analyzing existing implementations. Web resources $\mathcal{W}$ enable dynamic retrieval of cited papers, missing components, and implementation details referenced but not fully specified in the target paper. Finally, the repository $\mathcal{X}$ provides code templates and reference implementations (in-context examples) that serve as bases for synthesis.

3.2. Proposed Multi-Stage Framework

NERFIFY has four stages that transform research papers into executable code. Each stage addresses specific challenges in automated code synthesis for complex vision systems, from paper understanding through visual quality refinement.

Stage 1: CFG Formalization and In-Context Learning

A foundation of NERFIFY is constructing the domain knowledge base $\mathcal{K}$ by formalizing Nerfstudio's architectural patterns as a context-free grammar (CFG) that constrains code generation. We curate pairs $\{(\mathcal{P}_i, \mathcal{C}_i)\}_{i=1}^{m}$ of NeRF papers and their corresponding implementations, which populate

$\mathcal{K}$ and serve as in-context examples $\mathcal{X}$ for our synthesis pipeline. The extraction function $\mathcal{E}(\cdot)$ employs MinerU [36], a state-of-the-art PDF-to-markdown conversion tool.

As shown in Figure 2, the paper summarization agent processes each document through multiple refinement stages. First, MinerU converts the PDF into a structured markdown representation preserving equations, tables, figures, references and implementation details. A subsequent cleaning agent then distills this raw conversion by removing irrelevant sections such as extended introductions, related work discussions and redundant references. This agent operates under strict preservation constraints: all equations, implementation pseudocode, architectural diagrams, and citation relationships must remain intact. The agent validates completeness by ensuring that key technical components identified in the abstract appear in the refined document. Each processed paper and its corresponding Nerfstudio implementation are then stored as structured examples in our knowledge base, forming the grammatical foundation that guides subsequent synthesis. Full details of the CFG formalization are provided in the supplementary material.

Stage 2: Compositional Dependency Resolution

NeRF papers are inherently compositional – for instance, ZipNeRF implementation requires the proposal network from Mip-NeRF 360, hash encoding from Instant-NGP, and anti-aliasing techniques that span three papers. Naïve approaches that retrieve only the target paper fail because critical implementation details reside in dependencies. Our paper extractor agent constructs a citation dependency graph $G' = (V', E')$ of all referenced papers and their transitive dependencies, with nodes $v \in V'$ representing papers and edges $(u, v) \in E'$ indicating that v depends on technical components from u. Given a target paper P_{target}, our agent performs iterative multi-hop retrieval through four steps:

1. Dependency discovery: Parse P_{target} to extract cited papers $\mathcal{C} = \{c_1, \ldots, c_n\}$ and identify which components are borrowed (e.g., "we use the proposal network from [3]").

2. Recursive resolution: For each $c_i \in \mathcal{C}$, retrieve:

$$\text{Dependencies}(c_i) = \{c_i\} \cup \bigcup_{d \in \text{cited}(c_i)} \text{Dependencies}(d),$$

where the recursion terminates when all components required to implement the target paper have been located.

3. Component extraction: For each paper in the transitive closure, extract specific components mentioned in P_{target} or required by intermediate dependencies. Specialized LLM agents identify (a) architectural modules: e.g., "hash encoding", (b) loss functions: e.g., "distortion loss", (c) training protocols: e.g., "proposal sampling with stop-gradient".

4. Termination criterion: Stop when the agent has retrieved all components needed to satisfy the interface contracts of P_{target} and no unresolved dependencies remain in the graph.

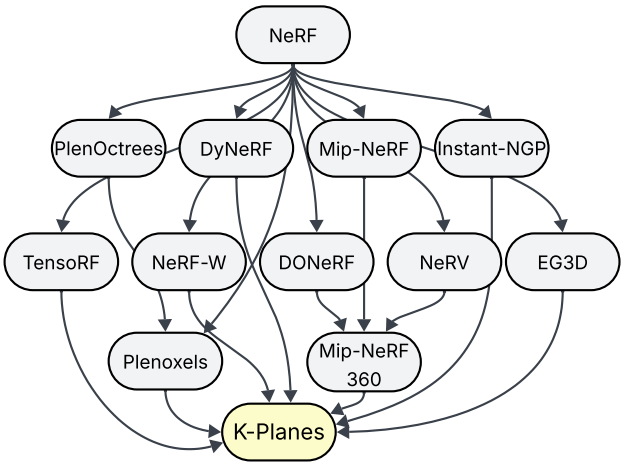

Figure 3. **NeRF citation dependency graphs.** Implementing K-Planes requires retrieving components from 7 direct dependencies (Plenoxels, TensoRF, Instant-NGP, Mip-NeRF 360, DyNeRF, EG3D, NeRF-W) and 12 total papers with transitive dependencies. Our compositional citation recovery automatically traverses such graphs to identify and retrieve all necessary components.

Figure 3 illustrates this process through K-Planes, which depends on 7 different papers. Our agent:
1. Identifies direct citations: Plenoxels (optimization), TensoRF (factorization), Instant-NGP (hash grids), Mip-NeRF 360 (proposal networks), DyNeRF (temporal sampling), EG3D (triplanes), NeRF-W (appearance codes).
2. Recursively retrieves dependencies: Mip-NeRF 360 requires Mip-NeRF, DONeRF, and NeRV; Plenoxels requires PlenOctrees; each ultimately traces back to NeRF.
3. Extracts components: proposal network and distortion loss from Mip-NeRF 360, hash encoder from Instant-NGP and VM decomposition from TensoRF.
4. Terminates: when all papers are processed and K-Planes' planar factorization $f(\mathbf{q}) = \prod_{c \in \mathcal{C}} f(\mathbf{q})_c$ can be implemented using the retrieved components.

Stage 3: Grammar-Guided Repository Generation

The master synthesis agent orchestrates code generation through a Graph-of-Thought (GoT) approach that ensures trainable code through syntax and execution feedback loops. It captures the tight inter-file couplings inherent in NeRF pipelines, where configurations flow to data managers, which feed fields, models, and ultimately the training pipeline.

As illustrated in Figure 4, the GoT code synthesis proceeds in four phases. During *DAG construction*, the parsed paper is mapped to a dependency graph over core Nerfstudio components. *Interface freezing* follows, where agents establish minimal public APIs in topological order. The *implementation* phase sees each node synthesize executable code validated through local contracts. *Integration testing* completes the cycle with end-to-end smoke tests that trigger

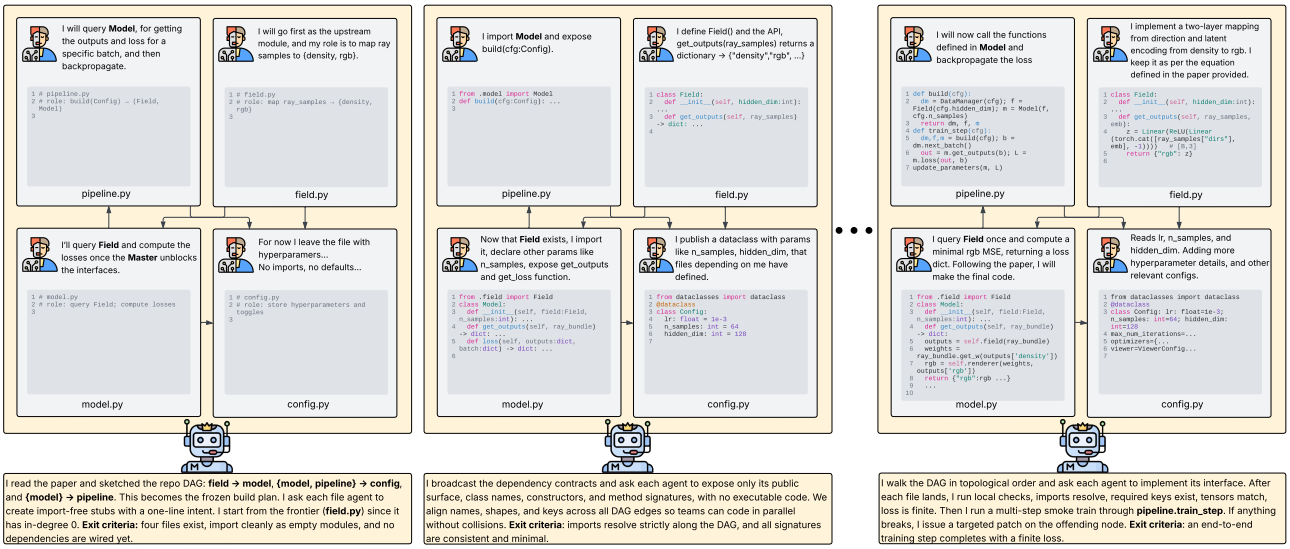

Figure 4. **Graph-of-Thought (GoT) Multi-Agent Code Synthesis**. The master agent orchestrates specialized file-agents that progressively build a NeRF repository over k steps. Each step shows files being created or modified through four stages: (1) DAG Construction maps papers to Nerfstudio component dependencies, (2) Interface Freeze establishes API contracts in topological order, (3) Implementation generates validated code with shape/gradient checks, (4) Integration Testing runs smoke tests with automated repair. Files evolve from minimal interfaces to complete implementations as agents coordinate through the dependency graph, producing runnable NeRF plugins.

automated critique and repair loops when failures occur.

The syntax and execution feedback loop iteratively refines code until all contracts pass, ensuring every generated repository can successfully train. This graph-native approach enables component-level fault localization, yielding faster convergence compared to monolithic code generation.

Stage 4: Visual-Driven Feedback

The final stage employs visual feedback to iteratively improve implementation quality beyond mere executability. After synthesis produces trainable code, we perform smoke training for 3k iterations (as used in our experiments). The system renders images from multiple camera viewpoints and dispatches them to our critique agent for analysis on three branches. The *metric branch* constructs dense error fields by computing local-window PSNR and SSIM maps, identifying regions of highest error through morphological operations. The *geometry branch* implements Cross-View Artifact Consensus, highlighting view-inconsistent structures indicative of floaters and ghosting. The *semantics branch* leverages Qwen3 VLM to analyze artifact triplets, to output structured diagnoses with candidate patches.

These three feedback mechanisms converge through the master agent, which applies patches within designated regions while maintaining revertability. The framework optionally allows user-driven feedback for domain experts to guide refinement (not used in our experiments for fair evaluations). The refinement loop continues until either: (1) the critique agent produces no further feedback, (2) the maximum iteration count is reached, or (3) the implementation achieves the PSNR targets reported in the original paper. This visual-driven approach consistently converges to within 0.5 dB PSNR and 0.2 SSIM of expert implementations.

4. Experiments

This section evaluates NERFIFY through comprehensive experiments on NERFIFY-BENCH our specialized benchmark for NeRF paper-to-code synthesis. We first introduce the benchmark composition and baseline systems, then present detailed evaluations of synthesis quality, novelty preservation, and component contributions.

4.1. NERFIFY-BENCH

To comprehensively evaluate our NERFIFY framework, we introduce NERFIFY-BENCH the first specialized benchmark for assessing paper-to-code systems that synthesize trainable NeRF code directly from research papers. While existing benchmarks such as Paper2CodeBench, RexBench, PaperBench, and Code-Dev evaluate general paper-to-code generation, they lack the domain-specific evaluation criteria necessary for NeRF implementations.

4.1.1. Benchmark Composition

NERFIFY-BENCH comprises **30 carefully selected papers**, classified across four evaluation categories to ensure comprehensive coverage of different implementation challenges:

Set 1. 10 Never-Implemented Papers: Papers without any publicly available source code, with expert-created reference implementations for evaluation. These papers are specifically chosen to avoid LLM training data contamination, since no public implementations exist, the LLMs used in our agents could not have seen corresponding code during pretraining, enabling unbiased evaluation of purely paper-driven synthesis capabilities.[1, 8, 19, 38, 46, 53, 54, 60, 62, 66].

Set 2. 5 Non-Nerfstudio Papers: Papers with existing public implementations but not integrated into the Nerfstudio, allowing direct comparison between generated code and original author implementations. [12, 20, 24, 28, 34]

Set 3. 5 Nerfstudio-Integrated Papers: Papers already integrated into Nerfstudio, serving as gold-standard references for evaluating synthesis quality. [23, 32, 33, 47, 55]

Set 4. 10 Novelty-Coverage Papers: Papers specifically selected for their distinct technical contributions (novel loss functions, architectural innovations, or training strategies) to evaluate how well our system captures and implements key research innovations. [2, 5, 9, 13, 14, 21, 22, 39, 48, 59]

4.2. Baseline Systems and Evaluation Setup

Baseline Systems. We evaluate NERFIFY against multiple baseline categories spanning generic to specialized approaches. *Paper2Code* and *AutoP2C* represent generic paper-to-code systems that prioritize breadth across all ML domains over domain-specific accuracy. For LLM-based approaches, *GPT-5* (current SOTA) employs single-shot code generation, lacking iterative planning or repairs, while *R1* employs retrieval-augmented in-context learning. We also compare against general multi-agent coding frameworks including *MetaGPT*, *ChatDev*, and *DeepCode*, which operate as software development agents without NeRF-specific architectural understanding or planning. As our gold standard, we include *Expert implementation* by researchers with NeRF expertise. Against these baselines, *Nerfify* employs the complete multi-agent pipeline with Graph-of-Thought synthesis, compositional citation recovery, and closed-loop refinement specifically designed for NeRF implementations.

Computational Setup. All experiments are conducted on NVIDIA A6000 GPUs with 48GB memory under consistent configurations. We train scenes for 100k iterations across standard benchmarks including Blender and DTU datasets.

Evaluation Overview. Our evaluation examines NERFIFY from multiple complementary perspectives. Section 4.3 compares generated code against expert implementations for papers without public code, measuring visual quality (PSNR, SSIM, LPIPS) vs what they have mentioned in the paper. Section 4.4 evaluates novelty preservation by analyzing whether systems correctly implement paper-specific innovations including novel equations, loss terms, and architectural components. Section 4.5 systematically ablates each system component to quantify individual contributions.

Paper	Reported			Human Impl.			NERFIFY (Ours)		
	PSNR↑	SSIM↑	LPIPS↓	PSNR↑	SSIM↑	LPIPS↓	PSNR↑	SSIM↑	LPIPS↓
KeyNeRF [38]	25.65	0.89	0.11	25.70	0.89	0.12	26.12	0.90	0.09
mi-MLP NerF [66]	24.70	0.89	0.09	22.64	0.87	0.15	22.85	0.87	0.15
ERS [46]	27.85	0.94	0.06	26.87	0.90	0.12	27.02	0.90	0.12
TVNeRF [62]	27.44	0.93	0.08	26.81	0.92	0.12	27.30	0.92	0.10

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.

This comprehensive framework reveals that domain-specific specialization in NERFIFY dramatically improves both executability (100% vs 5% for baselines) and algorithmic fidelity compared to generic approaches.

4.3. Comparison to Expert Implementations

Evaluation Metrics. We evaluate visual quality using three metrics: PSNR measures pixel-level reconstruction accuracy, SSIM captures perceptual similarity, and LPIPS quantifies perceptual distance using deep features. We report results on standardized test views from paper-specified datasets.

4.3.1. Set 1: Never-Implemented Papers

We first compare against expert implementations for NeRF papers without public code, where no code contamination is possible within the LLM's pretrained knowledge, ensuring unbiased assessment of true paper-to-code synthesis capabilities. Expert developers required 1-2 weeks per paper to create reference implementations, while NERFIFY generates comparable code in minutes.

Quantitative Results. Table 1 presents quantitative comparisons for representative papers in Set 1. Each paper is evaluated following its paper experiments: KeyNeRF and mi-MLP NeRF report averaged metrics across eight Synthetic-NeRF scenes, ERS uses the DTU dataset, while TVNeRF provides results for a single scene of Synthetic-NeRF(hotdog). NERFIFY achieves visual quality matching expert implementations within 0.5 dB PSNR and 0.02 SSIM on average. Complete results are in the supplementary.

In contrast, all baselines often fail to produce executable code as shown in Table 2. These failures occur because generic systems cannot recover implicit dependencies from citations or properly structure multi-file architecture. While baselines may generate syntactically valid Python, they lack the domain knowledge to wire components correctly or implement precise mathematical formulations.

Qualitative Comparison. Figure 5 shows qualitative comparisons on novel viewpoints. NERFIFY reproduces fine details including specular highlights, geometric edges, and texture patterns, validating that it captures the visual quality and improvements described in papers.

Metric	Paper2Code	AutoP2C	GPT-5	R1	Nerfify
Imports Resolve	✓	×	✓	✓	✓
Compiles/Trainable	×	×	×	×	✓
Training Stability	×	×	×	×	✓
Converges to Paper Results	×	×	×	×	✓

Table 2. **Comparison of NERFIFY with baselines in terms of executable code.** We evaluate ability to produce functional, trainable implementations. All baselines fail to generate trainable code despite some producing syntactically valid Python.

Method	Original Repository			Nerfify		
	PSNR ↑	SSIM ↑	LPIPS ↓	PSNR ↑	SSIM ↑	LPIPS ↓
Vanilla NeRF [32]	31.36	0.95	0.04	31.36	0.95	0.04
Nerfacto [47]	20.36	0.82	0.22	20.36	0.82	0.22
ℓ_0 Sampler [24]	29.21	-	0.04	30.13	0.97	0.03
InfoNeRF [20]	18.27	0.81	0.23	17.87	0.69	0.44

Table 3. **Comparison with existing implementations**. Evaluation of NERFIFY against original author repositories or gold-standard implementations (ℓ_0 Sampler doesn't report SSIM).

4.3.2. Sets 2, 3: Papers with Existing Implementations

Table 3 shows comparisons between NERFIFY and existing implementations for four papers from Sets 2 and 3 with public code – either gold-standard Nerfstudio repositories [32, 47] or original author-provided implementations [20, 24]. NERFIFY achieves comparable performance to official implementations with automated Nerfstudio integration. We note that for these papers, LLMs may have encountered their codebases during pretraining. Thereby, NERFIFY yields exactly the same code as the Nerfstudio repositories for [32, 47]. For papers with non-standard author-provided implementations [20, 24], NERFIFY results in standardized Nerfstudio-compatible code that yields comparable performance. Additional results are shown in supplementary.

4.4. Novelty Coverage Analysis

We evaluate whether NERFIFY and baseline methods faithfully implement paper-specific innovations across 10 complex NeRF papers from Set 4 of NERFIFY-BENCHFor each paper, we identify all novelty items including equations, loss terms, architectural blocks, training schedules, etc.

Evaluation. Let $\mathcal{N}$ denote the set of novel components identified in paper P. We compute four complementary metrics: C measures the fraction of components with *correct* implementation, I captures *incomplete* but partially correct implementations, M for *missing* unimplemented components, and $W = |\{n \in \mathcal{N} : |\theta_n - \hat{\theta}_n| < 0.1|\theta_n|\}|/|\mathcal{N}|$ evaluates fidelity of *weights* where θ_n and $\hat{\theta}_n$ represent paper-specified and implemented values, respectively. Note that NERFIFY can adaptively modify $\hat{\theta}_n$ through its Visual-Driven Feedback to achieve better training dynamics than the original paper specifications, distinguishing it from baselines that attempt only exact reproduction. To capture semantic understanding beyond structural metrics, we employ an LLM-based *score* that assesses implementation fidelity as $(\sum_{i=1}^{n} w_i \cdot s_i)/\sum_{i=1}^{n} w_i$, where $w_i \in [0, 1]$ represents the importance of novelty i extracted from paper, and $s_i \in \{0, 0.2, 0.4, 0.6, 0.8, 1.0\}$ indicates implementation completeness on a 6-level scale ranging from unimplemented to fully correct (details in supplementary). The LLM score analyzes both mathematical formulations and algorithmic logic, recognizing equivalent implementations and scoring each novelty component and its weight.

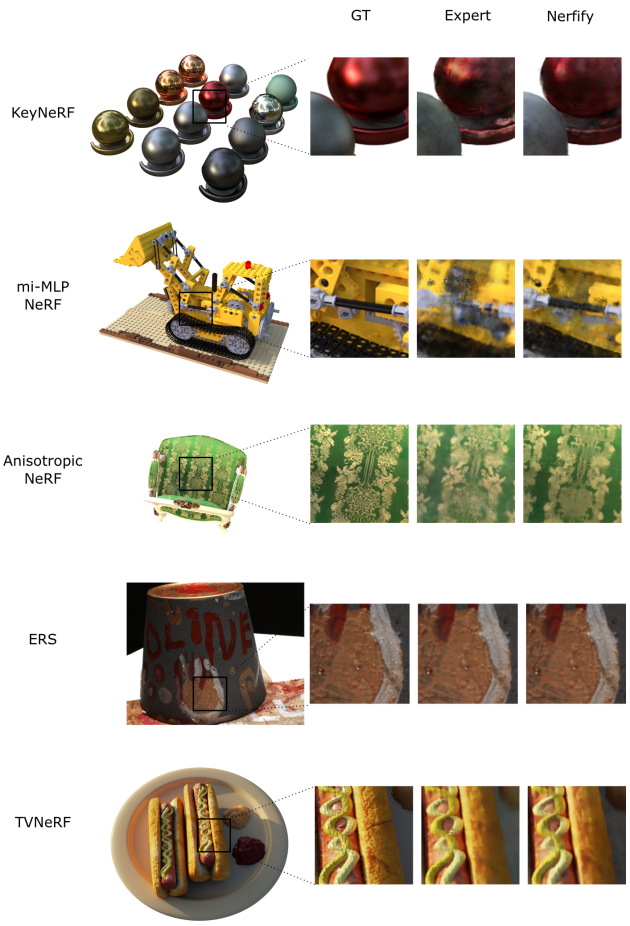

Figure 5. **Visual Comparison of NERFIFY and Human Implementation**. **Left**: Ground Truth Image, **Middle**: Expert Implementation, **Right**: Agent Implementation.

Results. As shown in Table 4, NERFIFY consistently achieves higher correct implementation rates across all evaluated papers compared to baselines. NERFIFY achieves perfect or near-perfect scores (C=1.00, M=0.00) across all papers, while generic baselines show significant component omissions and lower implementation fidelity. We note that achieving competent code generation from baseline LLM-based systems typically requires expert-level prompt engineering, which NERFIFY does not require. Complete paper-

Paper	Paper2Code					AutoP2C					R1					GPT-5					Nerfify				
	C↑	I↓	M↓	W↑	Score↑	C↑	I↓	M↓	W↑	Score↑	C↑	I↓	M↓	W↑	Score↑	C↑	I↓	M↓	W↑	Score↑	C↑	I↓	M↓	W↑	Score↑
Mip-NeRF [2]	0.83	0.17	0.00	0.83	0.85	0.17	0.17	0.66	0.25	0.20	0.67	0.17	0.16	0.67	0.58	0.50	0.30	0.20	0.60	0.58	1.00	0.00	0.00	1.00	1.00
BioNeRF [39]	0.30	0.40	0.30	0.40	0.35	0.10	0.30	0.60	0.10	0.15	0.70	0.20	0.10	0.70	0.75	0.80	0.10	0.10	0.80	0.82	1.00	0.00	0.00	1.00	1.00
PyNeRF [48]	0.50	0.30	0.20	0.60	0.58	0.00	0.10	0.90	0.10	0.03	0.30	0.60	0.10	0.70	0.68	0.40	0.30	0.30	0.80	0.52	1.00	0.00	0.00	0.90	0.97
TensoRF [5]	0.20	0.30	0.50	0.30	0.12	0.10	0.20	0.70	0.15	0.28	0.60	0.20	0.20	0.70	0.65	0.70	0.10	0.20	0.75	0.72	1.00	0.00	0.00	0.95	0.98
Tetra-NeRF [22]	0.13	0.25	0.63	0.20	0.22	0.00	0.13	0.88	0.00	0.08	0.63	0.25	0.13	0.70	0.72	0.50	0.25	0.25	0.60	0.58	1.00	0.00	0.00	1.00	1.00
E-NeRF [21]	0.38	0.25	0.38	0.60	0.48	0.00	0.13	0.88	0.00	0.05	0.63	0.25	0.13	0.80	0.72	0.50	0.25	0.25	0.75	0.60	1.00	0.00	0.00	0.95	1.00
StyleNeRF [13]	0.30	0.40	0.30	0.46	0.28	0.00	0.10	0.90	0.00	0.00	0.50	0.30	0.20	0.64	0.62	0.40	0.30	0.30	0.55	0.52	1.00	0.00	0.00	1.00	0.98
iNeRF [59]	0.70	0.20	0.10	0.80	0.75	0.00	0.10	0.90	0.00	0.05	0.60	0.30	0.10	0.70	0.68	0.50	0.30	0.20	0.60	0.58	1.00	0.00	0.00	1.00	0.97
SigNeRF [9]	0.38	0.38	0.24	0.50	0.52	0.00	0.13	0.87	0.00	0.08	0.63	0.25	0.12	0.75	0.72	0.50	0.25	0.25	0.63	0.58	1.00	0.00	0.00	1.00	1.00
MCNeRF [14]	0.00	0.13	0.88	0.20	0.15	0.00	0.25	0.75	0.10	0.08	0.50	0.38	0.13	0.80	0.74	0.75	0.25	0.00	0.85	0.95	1.00	0.00	0.00	1.00	0.95

Table 4. **Novelty coverage analysis across NERFIFY-BENCH papers**. For each baseline system, we report: C (Correct implementation rate), I (Incorrect/partial implementation rate), M (Missing component rate), W (Hyperparameter weight match accuracy), and $\text{Score}_{\text{LLM}}$ (overall semantic implementation score on 0-1 scale). NERFIFY achieves perfect or near-perfect scores (C=1.00, M=0.00) across all papers, while generic baselines show significant component omissions (M=0.12-0.90 for most methods) and lower implementation fidelity. Metrics are computed over all novel components identified in each paper, with weights derived from paper emphasis and experimental validation.

Configuration	Score	Trainable (%)	Correct Novelties Impl. (C)
NERFIFY (Full)	0.98	100	1.00
Knowledge Sources:			
w/o In-context Examples (Stage 1)	0.71	90	1.00
w/o citation recovery (Stage 2)	0.68	100	0.65
w/o Both	0.58	90	0.65
Validation & Feedback:			
w/o Smoke Tests (Stage 3)	0.69	60	0.85
w/o VLM Feedback (Stage 4)	0.99	100	1.00
Planning Strategy:			
One-Shot (no GoT) (Stage 3)	0.45	70	1.00

Table 5. **Component ablation study**. We evaluate the impact of each system component on synthesis quality and efficiency. Numbers are averaged over 10 papers from NERFIFY-BENCH.

to-code snippet comparisons for all identified novelties and additional comparison with code-generation tools like ChatDev [40], MetaGPT [17], and DeepCode [15] are provided in the supplementary material.

4.5. Ablation Study

In Table 5, we selectively disable key elements of NERFIFY (other metrics like PSNR are in supplementary). The results validate our multi-agent architecture, demonstrating the importance of domain knowledge, compositional reasoning, iterative validation, and visual refinement.

Knowledge Sources. Removing in-context examples drops score from 0.98 to 0.71 while trainability falls to 90%, though novelty implementation remains perfect (C=1.00). This indicates agents can interpret equations correctly but struggle with architectural integration. Citation dependencies proves equally critical: without it, the system maintains trainability but fails to implement 35% of novel techniques (C=0.65). Disabling both reduces semantic score to 0.58 while trainability remains at 90%, confirming that code quality degradation stems primarily from improper architectural patterns rather than runtime failures.

Validation and Feedback. Smoke tests significantly impact trainability. Without incremental validation, trainability drops to 60% and correct novelty implementation falls to 0.85 as interface mismatches accumulate during generation. VLM-guided feedback maintains perfect implementation (C=1.00) but slightly reduces semantic score from 0.99 to 0.98, reflecting hyperparameter adjustments that prioritize practical convergence over strict paper fidelity.

Planning Strategy. GoT synthesis substantially improves code quality over one-shot generation. While one-shot maintains perfect novelty implementation (C=1.00) and 70% trainability, its semantic score collapses to 0.45, indicating failures in establishing proper module boundaries and abstractions despite correct equation implementation.

5. Conclusion

We presented NERFIFY, a multi-agent framework that enables paper-to-code synthesis for fully trainable NeRF implementations through domain-aware reasoning and structured code generation. By formalizing Nerfstudio as a context-free grammar and orchestrating code generation through graph-of-thought planning, the framework ensures architectural correctness and full executability across complex NeRF pipelines. Its compositional citation recovery and visual-driven feedback enable agents to reconstruct and refine hidden dependencies, achieving expert-level visual quality within 0.5 dB PSNR while reducing development time from weeks to minutes. Beyond accelerating NeRF research, NERFIFY highlights how depth of specialization, rather than model scale alone, enables reliable translation of scientific ideas into working code. This insight suggests a template where domain-specific grammars and agentic reasoning can transform research reproducibility in other communities too. Future work will extend NERFIFY to further NeRF variants and NeRF-based methods, other areas of computer vision research and broader paper-to-experiment frameworks.

References

[1] Relja Arandjelović and Andrew Zisserman. Nerf in detail: Learning to sample for view synthesis. *arXiv preprint arXiv:2106.05264*, 2021. 6

[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. 6, 8

[3] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5470–5479, 2022. 2, 4

[4] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, et al. Graph of thoughts: Solving elaborate problems with large language models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38:17682–17690, 2024. 2

[5] 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. 6, 8

[6] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, 2024. 2

[7] Cognition Labs. Devin: The first ai software engineer. https://www.cognition-labs.com/introducing-devin, 2024. 2

[8] Congyue Deng, Jiawei Yang, Leonidas Guibas, and Yue Wang. Rethinking directional integration in neural radiance fields. *arXiv preprint arXiv:2311.16504*, 2023. 6

[9] 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. 6, 8

[10] Sara Fridovich-Keil, Giacomo Meanti, Frederik Rahbæk Warburg, Benjamin Recht, and Angjoo Kanazawa. K-planes: Explicit radiance fields in space, time, and appearance. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12479–12488, 2023. 2

[11] Shubham Gandhi, Dhruv Shah, Manasi Patwardhan, Lovekesh Vig, and Gautam Shroff. Researchcodeagent: An llm multi-agent system for automated codification of research methodologies. In *International Workshop on AI for Transportation*, pages 3–37. Springer, 2025. 2

[12] 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. 6

[13] 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. 6, 8

[14] 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. 6, 8

[15] HKUDS. Deepcode: Open agentic coding. https://github.com/HKUDS/DeepCode, 2025. GitHub repository. 8

[16] Samuel Holt, Max Ruiz Luyten, and Mihaela van der Schaar. L2mac: Large language model automatic computer for extensive code generation. *arXiv preprint arXiv:2310.02003*, 2023. 2

[17] 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. 2, 8

[18] Dong Huang, Qingwen Bu, Jie Zhang, Michael Xie, et al. Agentcoder: Multi-agent code generation with effective testing and self-optimisation. *arXiv preprint arXiv:2312.13010*, 2024. 2

[19] Byeongin Joung, Byeong-Uk Lee, Jaesung Choe, Ukcheol Shin, Minjun Kang, Taeyeop Lee, In So Kweon, and Kuk-Jin Yoon. Stable surface regularization for fast few-shot nerf, 2024. 6

[20] 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. 6, 7

[21] Simon Klenk, Lukas Koestler, Davide Scaramuzza, and Daniel Cremers. E-nerf: Neural radiance fields from a moving event camera. *IEEE Robotics and Automation Letters*, 8 (3):1587–1594, 2023. 6, 8

[22] 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. 6, 8

[23] 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. 6

[24] 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. 6, 7

[25] Zijie Lin, Yiqing Shen, Qilin Cai, He Sun, Jinrui Zhou, and Mingjun Xiao. Autop2c: An llm-based agent framework for code repository generation from multimodal content in academic papers. *arXiv preprint arXiv:2504.20115*, 2025. 2

[26] Jane Luo, Xin Zhang, Steven Liu, Jie Wu, Jianfeng Liu, Yiming Huang, Yangyu Huang, Chengyu Yin, Ying Xin, Yuefeng Zhan, et al. Rpg: A repository planning graph for unified and scalable codebase generation. *arXiv preprint arXiv:2509.16198*, 2025. 2

[27] Yujie Luo, Zhuoyun Yu, Xuehai Wang, Yuqi Zhu, Ningyu Zhang, Lanning Wei, Lun Du, Da Zheng, and Huajun Chen. Executable knowledge graphs for replicating ai research. *arXiv preprint arXiv:2510.17795*, 2025. 2

[28] 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. 6

[29] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 2023. 2

[30] Angelos Mavrogiannis, Joseph Krebs, Yifan Chen, et al. Combining llm code generation with formal specifications and reactive program synthesis. *arXiv preprint arXiv:2410.19736*, 2024. 2

[31] Jiacheng Miao, Joe R Davis, Yaohui Zhang, Jonathan K Pritchard, and James Zou. Paper2agent: Reimagining research papers as interactive and reliable ai agents. *arXiv preprint arXiv:2509.06917*, 2025. 2

[32] 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, 6, 7

[33] 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. 2, 6

[34] 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. 6

[35] Ziyi Ni, Yifan Li, Ning Yang, Dou Shen, Pin Lv, and Daxiang Dong. Tree-of-code: A tree-structured exploring framework for end-to-end code generation and execution in complex task handling. *arXiv preprint arXiv:2412.15305*, 2024. 2

[36] 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. 4

[37] OpenAI Team. Paperbench: Evaluating ai's ability to replicate ai research. In *OpenReview*, 2025. 2

[38] 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. 6

[39] 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. 6, 8

[40] 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. 2, 8

[41] Edward Raff. A step toward quantifying independently reproducible machine learning research. In *Advances in Neural Information Processing Systems*, 2019. 1

[42] Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. Paper2code: Automating code generation from scientific papers in machine learning. *arXiv preprint arXiv:2504.17192*, 2025. 1, 2

[43] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2023. 2

[44] Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, et al. Paperbench: Evaluating ai's ability to replicate ai research. *arXiv preprint arXiv:2504.01848*, 2025. 1

[45] Hongqiu Sun, Yuhao Li, Jiachen Jia, Haoye Zhou, and Zhengwei Liu. Clover: Closed-loop verifiable code generation. *arXiv preprint arXiv:2310.17807*, 2024. 2

[46] 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. 6

[47] 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. 2, 6, 7

[48] Haithem Turki, Michael Zollhöfer, Christian Richardt, and Deva Ramanan. Pynerf: Pyramidal neural radiance fields. *Advances in neural information processing systems*, 36:37670–37681, 2023. 6, 8

[49] Ce Wang, Bei Chen, Yuxuan Zhang, and Xiaodong Xie. Codep: Grammatical seq2seq model for general-purpose code generation. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 456–468, 2023. 2

[50] Qian Wang. nerf-arxiv-daily. https://github.com/wangqiannudt/nerf-arxiv-daily, 2025. GitHub repository; accessed 2025-11-08. 1

[51] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.01030*, 2024. 2

[52] Xingyao Wang, Boxuan Hou, Yiming Fang, Fangwei Wu, Tianyi Wu, Qian Chen, Yanxin Li, Haoyu Qin, and Heng Ji. Opendevin: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024. 2

[53] Yifan Wang, Jun Xu, Y Zeng, and Y Gong. Anisotropic neural representation learning for high-quality neural rendering. *arXiv preprint arXiv:2311.18311*, 2023. 6

[54] 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. 6

[55] Frederik Warburg, Ethan Weber, Matthew Tancik, Aleksander Holynski, and Angjoo Kanazawa. Nerfbusters: Removing ghostly artifacts from casually captured nerfs. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 18120–18130, 2023. 6

[56] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang,

Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024. 2

[57] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024. 2

[58] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom L Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2023. 2

[59] 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. 6, 8

[60] 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. 6

[61] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. *arXiv preprint arXiv:2404.05427*, 2024. 2

[62] 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. 6

[63] Yunzhi Zhang, Zizhang Li, Matt Zhou, Shangzhe Wu, and Jiajun Wu. The scene language: Representing scenes with programs, words, and embeddings. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 24625–24634, 2025. 2

[64] Xuanle Zhao, Zilin Sang, Yuxuan Li, Qi Shi, Weilun Zhao, Shuo Wang, Duzhen Zhang, Xu Han, Zhiyuan Liu, and Maosong Sun. Autoreproduce: Automatic ai experiment reproduction with paper lineage. *arXiv preprint arXiv:2505.20662*, 2025. 2

[65] Mingyang Zhou, Quanming Yao, Lun Du, Lanning Wei, and Da Zheng. Reflective paper-to-code reproduction enabled by fine-grained verification. *arXiv preprint arXiv:2508.16671*, 2025. 2

[66] 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. 6