ALGOGEN: Tool-Generated Verifiable Traces for Reliable Algorithm Visualization

Anonymous ACL submission

Abstract

Algorithm Visualization (AV) helps students build mental models by animating algorithm execution states. Recent LLM-based systems such as CODE2VIDEO generate AV videos in an end-to-end manner. However, this paradigm requires the system to simultaneously simulate algorithm flow and satisfy video rendering constraints (element layout, color schemes, etc.), a complex task that induces LLM hallucinations. This results in reduced execution success rates, element overlap, and inter-frame inconsistencies. To address these challenges, we propose ALGOGEN, a novel paradigm that decouples algorithm execution from rendering. We first introduce Visualization Trace Algebra (VTA), a monoid over algorithm visual states and operations. The LLM then generates a Python tracker that simulates algorithm flow and outputs VTA-JSON traces, a JSON encoding of VTA. For rendering, we define a Rendering Style Language (RSL) to templatize algorithm layouts. A deterministic renderer then compiles algorithm traces with RSL into Manim, LaTeX/TikZ, or Three.js outputs[1]. Evaluated on a LeetCode AV benchmark of 200 tasks, ALGOGEN achieves an average success rate improvement of 17.3% compared to end-to-end methods (99.8% vs. 82.5%). These results demonstrate that our decoupling paradigm effectively mitigates LLM hallucinations in complex AV tasks, providing a more reliable solution for automated generation of high-quality algorithm visualizations. Demo videos and code are available at: `https://anonymous.4open.science/w/algogen-E227/`.

1 Introduction

Algorithm visualization (AV) (Shaffer et al., 2010; Naps et al., 2000) helps learners bridge the gap between abstract pseudocode and concrete mental models by animating execution states, with empirical studies showing improved learning outcomes (Naps et al., 2002; HUNDHAUSEN et al., 2002). Classical systems such as JHAVE (Naps et al., 2000) and VisuAlgo (Halim et al., 2012) provide textbook-style animations for sorting, graphs, and dynamic programming. However, creating high-quality AV remains costly: instructors must design example inputs, select states to highlight, implement visualization logic in specific toolkits (e.g., Manim (The Manim Community Developers, 2024)), and iterate on layout and aesthetics. This cost prevents scaling AV coverage to thousands of algorithm problems on online judges and MOOCs (Shaffer et al., 2010; Halim et al., 2012), even as students continue reporting difficulties with core concepts (Zingaro et al., 2018). Meanwhile, LLMs have demonstrated strong code generation capabilities (Chen et al., 2021) and now power educational tools like GPTutor (Chen et al., 2023a), raising a natural question: can LLMs automatically generate high-quality algorithm visualizations from problem descriptions at scale? While LLMs show promise in generating charts (Dibia, 2023), algorithm visualization presents unique challenges due to complex state dynamics, making this question both non-trivial and potentially transformative for democratizing algorithm teaching materials.

Recent systems such as TheoremExplainAgent (Ku et al., 2025) and CODE2VIDEO (Chen et al., 2025) prompt LLMs to produce AV content end-to-end: given an algorithm, the model directly outputs Manim code that is rendered into animation. However, this approach suffers from four limitations: (1) **Unverifiable correctness**—the pipeline produces only code-only Manim scripts (monolithic or multi-scene) without structured traces of intermediate states, making systematic verification difficult and forcing manual spot-checking; (2) **Layout instability**—LLMs place visual ele-

[1] Manim, TikZ, and Three.js are respectively a Python animation engine, a LaTeX vector graphics package, and a JavaScript 3D rendering library.

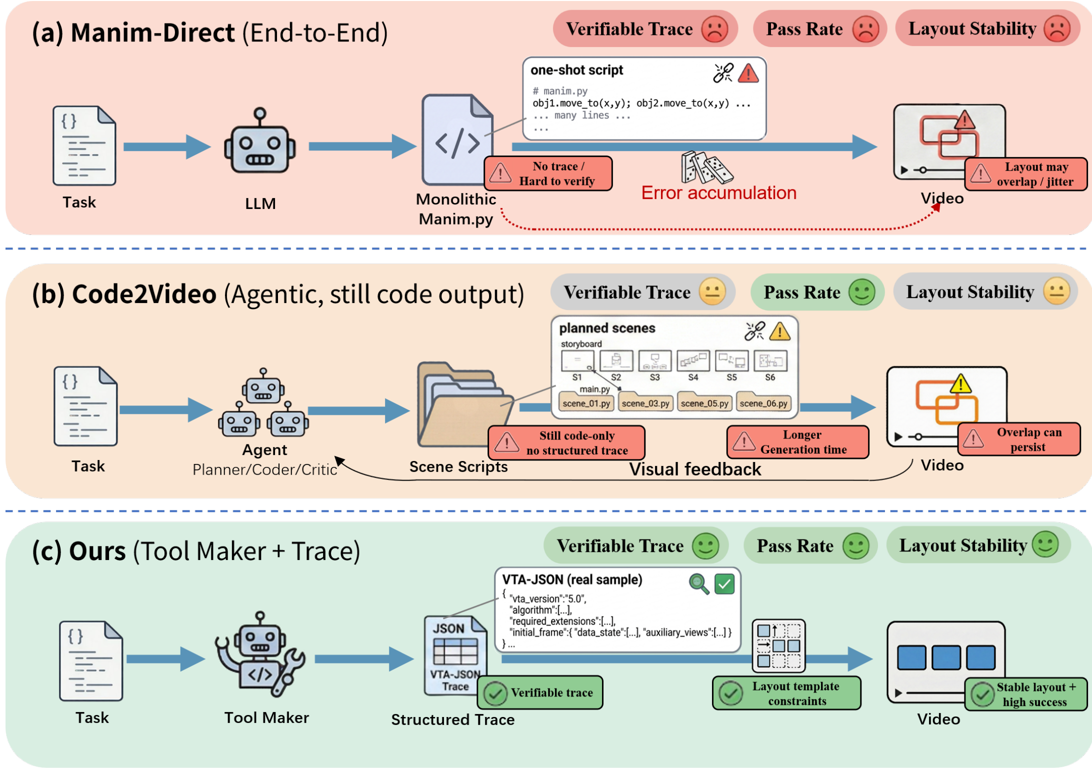

Figure 1: Paradigm comparison. (a) Manim-Direct outputs a monolithic Manim script (no traces → error accumulation and layout jitter/overlap). (b) Code2Video plans multiple `scene_*.py` files with visual feedback, but still code-only (no schema-validated trace) and slower. (c) Ours outputs an executable tracker + schema-validated VTA-JSON 5.0 trace, rendered under RSL templates for stable, verifiable videos.

ments with ad-hoc coordinates, causing overlaps and frame-to-frame jitter due to the lack of consistent canvas-level planning; (3) **Low reuse**—fixing errors requires regenerating entire scripts without reusable intermediate representations; and (4) **Latent execution complexity**—end-to-end generation forces LLMs to implicitly simulate execution without explicit chain-of-thought reasoning (Wei et al., 2023), exceeding their ability to jointly satisfy algorithmic correctness, rendering API calls, and layout constraints. On our 200-task LeetCode benchmark, the Manim-Direct baseline achieves only 82.5% rendering success, suggesting the core bottleneck is not lack of algorithmic knowledge but *task complexity*: end-to-end generation creates a long-horizon planning problem with large action spaces where small early deviations cascade across subsequent steps, preventing consistent generation of correct and readable scripts.

To address these challenges, we decompose AV generation into smaller sub-tasks—tracker generation, execution trace validation, and deterministic rendering—narrowing the action space so each step has clear, standardized outputs. Drawing on augmented language models (Mialon et al., 2023; Schick et al., 2023), Program-of-Thoughts (Chen et al., 2023b), and LLMs as tool makers (Cai et al., 2024), we propose the *LLM Tool Generation* paradigm: the LLM acts as a tool maker rather than direct video generator. First, the LLM generates a Python tracker that emits execution information in VTA-JSON 5.0, the JSON encoding of our Visualization Trace Algebra (VTA)—a monoid describing algorithmic visual states and composable operations with schema-validated correctness. Second, the model generates a Rendering Style Language (RSL) specification that selects safe layout templates and aesthetic styles. Finally, deterministic renderers interpret VTA under RSL guidance to produce Manim videos, LaTeX/TikZ figures (Tantau, 2025), or interactive Three.js (mrdoob and contributors, 2025) scenes.

This decomposition offers four advantages: (1) **Verifiability**—the tracker outputs schema-validated JSON traces testable on multiple inputs and comparable to reference implementations, enabling systematic automated correctness checking; (2) **Reusability**—a single tracker serves multiple inputs and rendering backends (Manim, TikZ, Three.js), amortizing LLM cost and ensuring con-

sistent visual design; (3) **Debuggability**—errors localize to specific trace steps with structured diagnostics for targeted retry, avoiding monolithic script editing; and (4) **Separation of concerns**—LLMs focus on algorithmic structure and pedagogical sequencing while layout and aesthetics are delegated to domain-specific algorithms and RSL-guided styling.

To rigorously evaluate our framework, we introduce ALGOGEN-Bench, a standardized benchmark derived from a large corpus of LeetCode problems collected via LeetCode's GraphQL endpoint. We convert problems into our unified task format and stratify-sample 200 tasks spanning six major algorithm families (Array, DP, Sorting, Graph, Tree, Hashtable) and multiple difficulty levels, with manual relabeling and test-case verification for reliability. Figure 3 summarizes the benchmark taxonomy, coverage, and difficulty composition. Overall, ALGOGEN-Bench enables reproducible evaluation of schema-validated, step-wise execution traces.

Our contributions are as follows:

1. We introduce a framework that decomposes AV generation into tracker generation, trace validation, and deterministic rendering, enabling verifiable correctness, cross-backend reusability, and explainable debugging.

2. We formalize a monoid-based algebra for algorithmic visual states and composable operations, encoded as schema-validated VTA-JSON 5.0, which decouples algorithmic semantics from rendering backends and enables automated correctness checking.

3. We design a Rendering Style Language that automates layout planning and aesthetic refinement, eliminating ad-hoc coordinates and ensuring collision-free compositions across Manim, LaTeX/TikZ, and Three.js.

4. We construct ALGOGEN-Bench with 200 diverse LeetCode problems, demonstrating 99.8% pipeline success and 99.2% algorithmic correctness vs. 82.5% and 87.0% for end-to-end baselines, establishing state-of-the-art for LLM-driven algorithm visualization.

2 Related Work

Classical Algorithm Visualization. Classical AV systems and program tracers like Python Tutor (Guo, 2013) demonstrate pedagogical benefits (Naps et al., 2002). Subsequent work has proposed large web-based platforms such as VisuAlgo and related unified environments for teaching data structures and algorithms (Halim et al., 2012), as well as dynamic, activity-focused AV systems and game-based visualizations (Vrachnos and Jimoyiannis, 2014; Su et al., 2021). However, these systems require substantial manual effort per algorithm, and coverage lags far behind the thousands of exercises on modern online judges (Zingaro et al., 2018). Recent platforms also integrate conversational agents with dynamic algorithm visualizations (e.g., VisualCodeMOOC) (Li et al., 2025).

LLM-Based Educational Content. LLMs have been applied to generate explanations and visualizations. TheoremExplainAgent (TEA) generates Manim videos for math proofs, and CODE2VIDEO explains code concepts via automatically generated animations (Ku et al., 2025; Chen et al., 2025). Beyond video generation, LLMs have also been deployed as interactive programming tutors and teaching assistants, for example GPTutor for line-by-line code explanation in VS Code (Chen et al., 2023a) and studies on using ChatGPT as a teaching assistant in data-structures-and-algorithms courses (Jamie et al., 2025). Unlike ALGOGEN, which uses a structured IR (VTA) to capture runtime state dynamics and ensures verifiability, these systems rely on end-to-end generation (TEA) or AST-based templates (CODE2VIDEO), limiting their ability to handle complex algorithm execution reliably.

LLM Code Generation. While LLMs excel at code generation, they often struggle with complex APIs (e.g., Manim) due to hallucinations (Zhong and Wang, 2024; Liu et al., 2023). Recent reviews and surveys highlight both rapid progress and brittleness when code LLMs are used in real-world development workflows (Husein et al., 2025; Jiang et al., 2025; Weber, 2024). Although retrieval-based methods such as DocPrompting (Zhou et al., 2023) help with library use, our analysis shows that adding RAG documentation often fails to fix API misuse *in practice*, especially under complex visual logic. More broadly, tool-using LLM agent frameworks improve reliability by delegating execution to external tools (Ge et al., 2023; Liu et al., 2024). LLM-based data visualization similarly combines IRs with deterministic rendering (Ouyang et al., 2025; Zhao et al., 2025). Instead, we have the LLM generate simple *tracker* code and shift complexity to deterministic renderers.

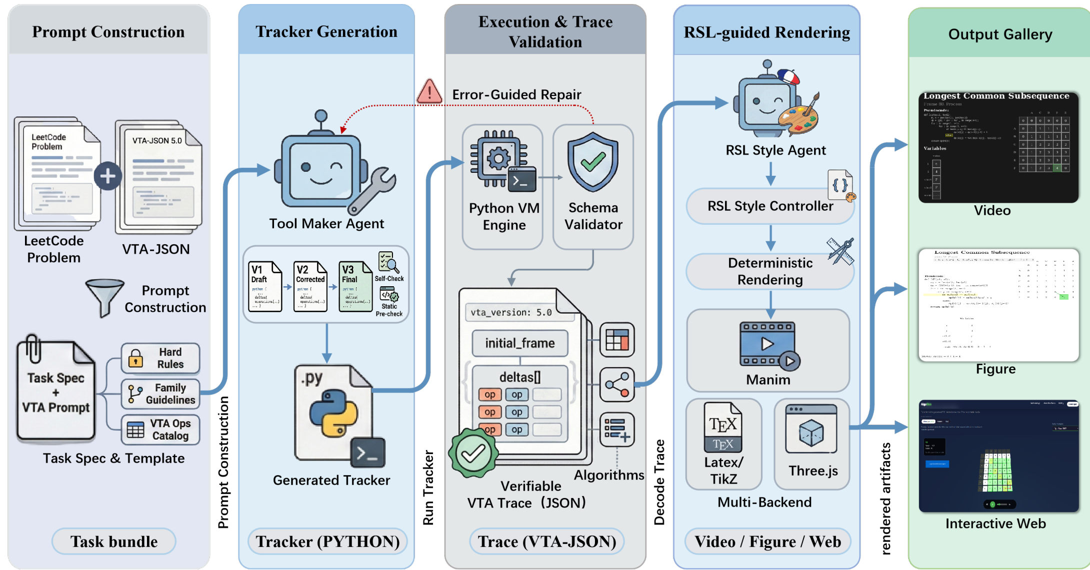

Figure 2: Overview of ALGOGEN: Prompt Construction → Tracker Generation → Execution & Trace Validation → RSL-guided Rendering. Validation failures trigger error-guided repair; RSL controls style/layout without changing trace semantics; deterministic backends include Manim, LaTeX/TikZ, and Three.js.

3 Method

3.1 Overview

Given an algorithm visualization task T (an algorithm problem description with auxiliary metadata), our goal is to produce a visualization artifact V (video/figure/interactive page). As shown in Figure 2, our key idea is to make the intermediate process controllable and verifiable: instead of synthesizing renderer-level animations end-to-end, we (i) represent executions as schema-validated VTA-JSON traces (VTA; Sec. 3.2), (ii) prompt an LLM to generate an executable Python tracker that emits these traces during execution (Sec. 3.3), and (iii) render validated traces with deterministic backends under an optional style specification in RSL (Sec. 3.4). This decomposition separates *what happens* (execution semantics) from *how it looks* (presentation), enabling validation and localized repair while keeping rendering deterministic.

3.2 VTA: Visualization Trace Algebra

Motivation and usage (I/O). We introduce VTA, a schema-validated VTA-JSON trace IR, because end-to-end renderer-level animation generation entangles long-horizon algorithm semantics with low-level rendering and layout decisions, making errors cascade across steps; VTA instead records execution as locally checkable, typed transitions, enabling modular verification/repair and deterministic replay across backends, and empirically improves reliability over end-to-end rendering in our experiments. Unlike declarative grammars for static visualization (e.g., Vega-Lite (Satyanarayan et al., 2017)), VTA directly models step-wise mutations (e.g., pointer movements and updates to auxiliary structures). In our pipeline, a Python tracker executes the algorithm and emits a trace: **input** is the tracker's runtime algorithm state (e.g., arrays/-graphs/DP tables and auxiliary variables) together with the current pseudocode line (for highlighting), and **output** is a schema-validated VTA-JSON file (`trace.json`) that deterministic renderers can consume to produce the final artifact.

Trace format and validation contract. An algorithm run is represented as a trace (s_0, w) and serialized in VTA-JSON: `initial_frame` encodes the initial visual state s_0, and `deltas` records the evolution as a sequence of small, typed operation batches. Each `delta` is aligned with a highlighted pseudocode line, and concatenating all deltas yields the full operation sequence w. We store traces as deltas (rather than absolute frames), aligning naturally with algorithm steps and animation primitives. A validator enforces VTA-JSON invariants—including version consistency, referential integrity, and type-correct operation parameters (Lu et al., 2025)—and traces must pass validation before deterministic rendering; otherwise, we trigger targeted error-guided repair using validator diagnostics. The state space is many-sorted (arrays, graphs, trees, hash tables, DP tables, and auxiliary views),

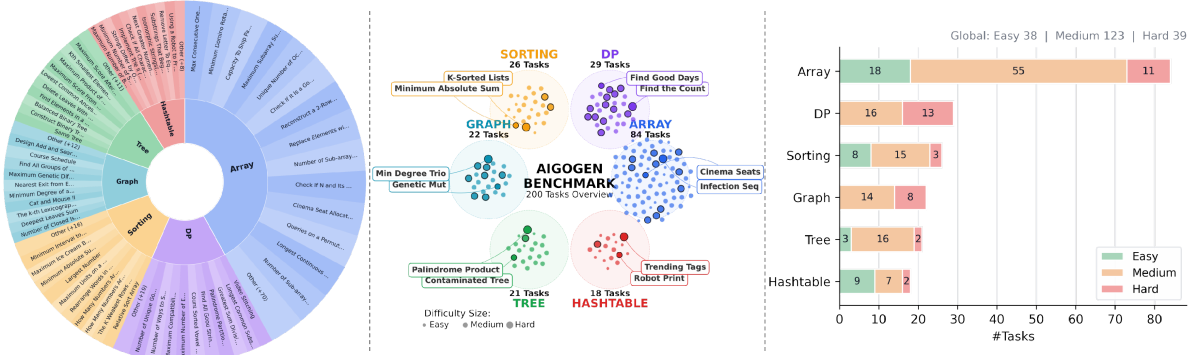

Figure 3: **Overview of ALGOGEN-Bench.** (1) **Left:** taxonomy of 200 LeetCode tasks across six algorithm families. (2) **Middle:** dataset coverage map; nodes denote tasks colored by family and sized by difficulty (Easy/Medium/Hard). (3) **Right:** difficulty breakdown per family (Global: 38 Easy, 123 Medium, 39 Hard).

and VTA provides a compact operation catalogue covering style updates, structural updates, and explanatory overlays (Appendix F).

Theoretical analysis. We model an algorithm visualization as a typed visual state $s \in S$ acted on by primitive operations. Each primitive operation symbol $o \in Op$ denotes a (partial) state transformer $[\![o]\!] : S \rightharpoonup S$. A finite sequence of operations $w = o_1 \dots o_n \in Op^*$ acts on s by composition:

$$s \cdot \epsilon = s, \qquad s \cdot (o_1 \dots o_n) = [\![o_n]\!] \circ \cdots \circ [\![o_1]\!](s),$$

where ϵ is the empty sequence. This view matches our trace format: each `delta` encodes a small word in Op^*, and a full VTA-JSON trace is obtained by concatenating these words. Deterministic renderers then interpret the trace by sequentially applying the corresponding state transformers.

Theorem 1. *Let Op be the primitive operation set in our* VTA-*JSON specification, and let $*$ be concatenation on Op^* with identity ϵ. Then $(Op^*, *, \epsilon)$ satisfies the monoid axioms:*

$$\begin{aligned} &\forall u, v, w \in Op^*, \quad (u * v) * w = u * (v * w), \\ &\forall u \in Op^*, \qquad \epsilon * u = u, \quad u * \epsilon = u. \end{aligned}$$

Where Op^* is the set of all finite sequences over Op (including ϵ). Proof of Theorem 1 in Appendix H. The monoid structure justifies treating delta batches as composable trace fragments: concatenation yields a well-defined trace independent of parenthesization. As a result, our pipeline can validate, debug, and render traces modularly while preserving execution semantics.

3.3 Tool Maker: LLM-Generated Trackers

Tracker Synthesizer. Because end-to-end renderer-level generation hides intermediate execution state and entangles algorithm semantics with low-level API/layout decisions, we instead use the LLM to generate a runnable tracker that executes the algorithm and emits a schema-validated trace. This module prompts the LLM to synthesize an instrumented Python tracker that emits VTA-JSON deltas, with execution/validation feedback enabling error-guided repair. **Input** is the Unified Task Bundle (task specification, test input, pseudocode, and the required VTA-JSON schema), and **output** is an executable `tracker.py` that produces a VTA-JSON trace (`trace.json`) when run. The tracker executes the algorithm on the task's input and emits VTA-JSON deltas via a lightweight `Visualizer` wrapper. The tracker is responsible for *what happens* during execution (i.e., correct state transitions and aligned highlights), while leaving *how it looks* to deterministic renderers.

To improve robustness, we use a three-stage generation strategy: (i) draft, (ii) self-refinement, and (iii) error-guided repair driven by execution errors and schema-validation feedback (Madaan et al., 2023). We also apply a lightweight static pre-check before execution. Full prompt templates and implementation details are provided in Appendix I (Wang et al., 2023).

3.4 Style Controller: RSL-Guided Deterministic Rendering

Style Controller. To avoid the layout instability and error accumulation of end-to-end script synthesis, we restrict the model to producing a high-level,

Method	Algorithm correctness
Ours (VTA+RSL)	**99.8**%
Ours (VTA-only)	99.2%
`manim_direct`	87.0%
`manim_direct_novta`	84.9%

Table 1: LLM-judged algorithm correctness for our systems and end-to-end Manim baselines.

schema-based style specification (RSL), while deterministic backends handle low-level graphics decisions when mapping validated traces to artifacts. Style Controller uses an LLM to specify high-level layout and aesthetics (RSL), while deterministic renderers map the validated trace to videos/figures/-pages. RSL is a compact JSON-based domain-specific language (DSL) that captures high-level layout and style preferences (e.g., `layout`, `theme`, `annotations`) and is generated once per task as a JSON script. Given lightweight trace metadata (e.g., algorithm family and trace length/#frames), the LLM generates an RSL instance conforming to a fixed schema; deterministic backends then validate and interpret this RSL into renderer-specific configuration, choosing suitable layouts and pacing without changing trace semantics. As a result, the model controls high-level appearance without issuing low-level graphics API calls. The RSL schema is given in Appendix G.

4 Experimental Setup

4.1 Dataset Construction

We construct a 200-task LeetCode benchmark with clear algorithmic logic, step-wise visualizable state changes, diverse difficulty (Easy/Medium/Hard), and coverage of six families (sorting, arrays, dynamic programming, trees, graphs, hash tables; Figure 3 shows the taxonomy view, coverage map, and per-family difficulty mix). Tasks are stored as standardized `example/*.txt` specifications; Appendix E details the format and quality-control procedures.

4.2 Models and Baselines

We use several LLMs as tool makers, including Qwen3-235B (Yang et al., 2025), DeepSeek V3.1 (DeepSeek-AI et al., 2025), and GLM-4.6 (GLM et al., 2024), accessed via the SiliconFlow API (SiliconFlow, 2025). We use the three-stage generation strategy in Section 3. For each task, we generate an initial tracker and allow up to

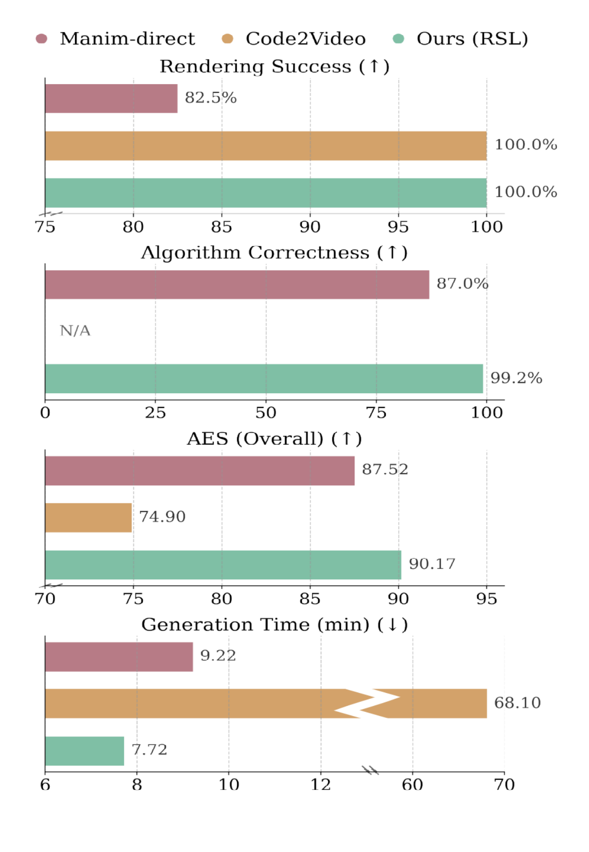

Figure 4: **System-level comparison across four dimensions.** Rendering success, algorithm correctness, aesthetic quality (AES), and generation time.

three error-guided repair attempts; the final tracker is reused across different inputs and backends.

We compare our VTA-based system with three end-to-end Manim baselines: **manim_direct**, where the LLM directly outputs Manim Scene code; **manim_direct_rag**, which adds 12k-token RAG-enhanced Manim documentation; and **manim_direct_novta**, a less structured baseline without VTA-style constraints (only loose prompts). All baselines share the same maximum tokens, and retry strategy as our method; full prompt templates and implementation details are provided in Appendix K.

Figure 5 presents a qualitative comparison on a representative dynamic-programming task (*Minimum Cost to Merge Stones*).

4.3 Evaluation Metrics

Rendering success rate. The fraction of tasks that complete tracker execution, trace validation, and rendering without unrecoverable exceptions (playable video).

System / Model	Metric	Success (# / 200)	Videos / Prompt size
DeepSeek-V3.1	Trace success (ours)	200 / 200 (100.0%)	200 AES/TEA videos
Qwen3-235B	Trace success (ours)	199 / 200 (99.5%)	199 AES/TEA videos
GLM-4.6	Trace success (ours)	200 / 200 (100.0%)	200 AES/TEA videos
Average	Trace success (ours)	599 / 600 (99.8%)	599/600 AES/TEA videos
Manim-Direct (No RAG)	Video rendering (e2e)	165 / 200 (82.5%)	165 videos, $\sim$ 4k tokens
Manim-Direct (RAG-enhanced)	Video rendering (e2e)	158 / 200 (79.0%)	158 videos, $\sim$ 12k tokens

Table 2: Success on the 200-task benchmark. Our VTA pipeline is evaluated by trace-generation success (valid VTA-JSON), while Manim-Direct baselines are evaluated by end-to-end video rendering success.

Method	#Eval	Efficiency (↓)		Aesthetics (↑)					
		Time (min)	Tokens (K)	EL	AT	LF	VC	AD	Avg.
Ours (VTA + RSL)	200	7.72	31.6	17.70	17.02	18.36	18.42	18.67	**90.17**
Ours (VTA Only)	599	8.59	20.3	17.36	16.67	18.01	18.03	18.37	88.45
manim_direct	156	9.22	17.4	17.35	16.48	17.93	17.90	17.87	87.52
manim_direct_novta	127	8.91	7.5	17.79	17.41	18.11	18.28	18.11	89.70
Code2Video	200	68.10	24.0	13.75	13.07	15.22	15.41	17.46	74.90

Table 3: AES comparison across systems, including the RSL-enhanced variant. Efficiency: Time (average minutes per task) and Tokens (average token consumption per topic). Aesthetics: Element Layout (EL), Attractiveness (AT), Logic Flow (LF), Visual Consistency (VC), Accuracy & Depth (AD), and Avg.

Algorithm correctness. Following LLM-as-judge practice (Zheng et al., 2023), a code-level evaluator scores each tracker with a rubric (algorithm logic, VTA compliance, presentation); we rescale the 0–50 algorithm-logic score to a 0–100 "algorithm-correctness" metric (Table 1).

Aesthetic quality (AES). We adopt the automatic aesthetic scoring model from CODE2VIDEO (Chen et al., 2025), following its AES rubric (5 dimensions) to rate frames and videos on a 0–100 scale. For each video, we report the average AES over sampled frames.

TEA evaluation. We additionally apply the multimodal evaluation rubric from TheoremExplainAgent (TEA) (Ku et al., 2025), using their 0–5 scale per dimension to assess visual and textual quality in a way comparable to prior work on educational video generation.

5 Results

5.1 Overall Performance

On the 200-task benchmark, our framework achieves state-of-the-art performance across rendering success, algorithm correctness, aesthetics, and efficiency, as summarized in Figure 4. Compared with the strongest end-to-end Manim baseline, our VTA-based tool-generation pipeline achieves substantially higher end-to-end success and correctness with lower failure rates; relative to the agentic `Code2Video` baseline, our RSL-guided renderer matches its rendering success while providing verifiable traces, higher AES scores, and significantly shorter generation time. Detailed metric values are reported in Tables 1, 2, and 3, with Appendix I, Table 9 further breaking down the rubric scores by dimension. Table 1 shows that VTA+RSL reaches 99.8% algorithm correctness (DeepSeek-V3.1), with 99.2% on average for VTA-only trackers, versus mid-80% for Manim baselines.

Across all families, tool-making consistently outperforms end-to-end script generation, with larger gains on long-horizon DP and graph/hashtable tasks (Table 10). Under our prompting and repair budget, the average LLM cost is ~\$0.022 per task (VTA+RSL, 200 tasks).

Analysis of the generation process shows that the error-guided repair mechanism is critical: the success rate without repair (Pass@1) averages 91.0% (e.g., 89.5% for Qwen3-235B), rising to 99.5% after up to 3 repair rounds. This demonstrates the efficacy of our compiler-like feedback loop (Madaan et al., 2023).

Adding 12k-token RAG documentation for Manim APIs actually *reduces* the baseline success rate from 82.5% to 79.0%, and over 80% of failures remain API misuse or logic drift. This suggests that the bottleneck is task formulation rather than library knowledge; detailed failure statistics are deferred to Appendix J.

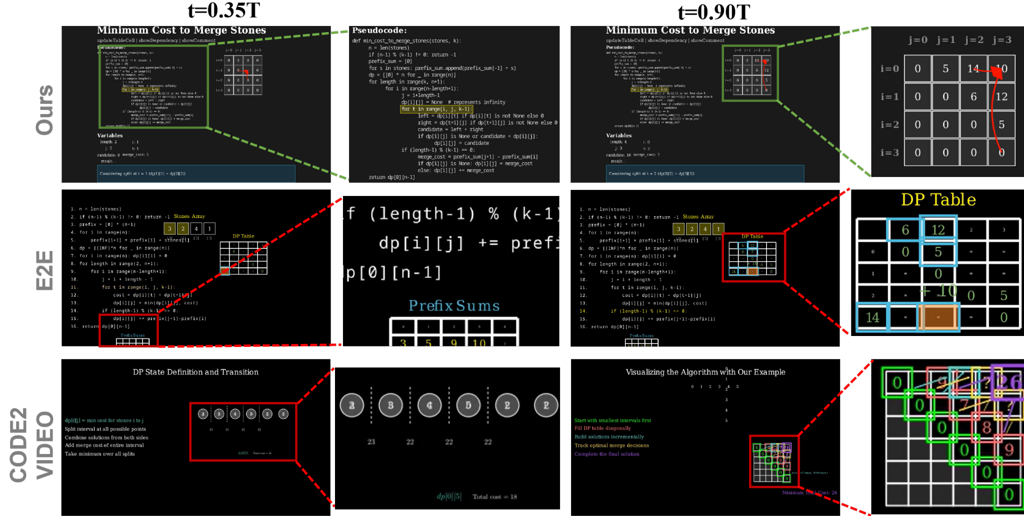

Figure 5: Qualitative comparison on a dynamic-programming task (*Minimum Cost to Merge Stones*).

5.2 Visual Quality: AES Evaluation

We use AES to evaluate videos across five 0–20 dimensions: layout, attractiveness, logic, accuracy & depth, and consistency. System-level comparisons are in Table 3; its time and token columns show that our VTA+RSL variant remains efficient while achieving higher AES scores than end-to-end Manim baselines. Per-model AES scores indicate that model choice has relatively minor impact on aesthetics (Appendix D, Table 4), while our VTA-only pipeline is already competitive with end-to-end Manim baselines. As an additional end-to-end reference, `code2video_manim_direct` achieves a substantially lower AES total of 74.90 on the 200 videos that were successfully evaluated. The RSL-enhanced variant further improves total AES to 90.17, mainly by better layout and color choices, without sacrificing correctness.

We also reuse TEA's multimodal evaluation rubric (Ku et al., 2025) as an auxiliary metric, consistent with the AES findings (full TEA results in Appendix D.2).

5.3 Case Study

Figure 7 shows examples from six algorithm families, demonstrating that our pipeline consistently visualizes core data structures and step-wise execution without occlusion or overlap. Array/DP examples highlight element updates and 2D table filling via synchronized cell highlighting. Graph/tree examples co-display structure with auxiliary states (e.g., queues/stacks), while hash-table examples show evolving key–value mappings (including bucket views). Overall, the main view and auxiliary panels remain spatially separated, making intermediate states easy to follow.

We observe a presentation limitation under high information density: long pseudocode blocks or large tables can force the canvas to rescale elements, reducing legibility (Figure 8). This affects layout/readability rather than semantic correctness, and could be mitigated by adaptive zooming, summarization, or paged/scrollable views.

6 Conclusion

We presented a unified framework for algorithm visualization that uses LLMs as *tool makers*. Instead of prompting the model to directly author renderer-level video scripts (e.g., Manim code), the model synthesizes an instrumented Python tracker that emits schema-validated VTA-JSON 5.0 traces, which deterministic engines then render into Manim videos, LaTeX figures, and Three.js interactions. We further introduced RSL to enable controlled, LLM-driven style adaptation without altering execution semantics.

On a 200-task LeetCode benchmark, our method significantly improves rendering success, algorithm correctness, and layout quality over strong end-to-end baselines. Beyond algorithm visualization, we believe the tool-maker paradigm can benefit other domains where reliability and controllability are critical, such as multi-tool AI agents and LLM-based data visualization systems, as well as broader UI automation and scientific simulation scenarios.

Limitations

Our work has several limitations.

First, the benchmark focuses on typical algorithmic tasks with relatively structured states (arrays, graphs, DP tables). More complex domains (e.g., geometric algorithms with continuous geometry or approximate methods) may require extending VTA with richer geometry support or probabilistic annotations.

Second, although we support multiple rendering backends, our empirical evaluation focuses primarily on Manim videos. Our experiments with TikZ figures and 3D interactions are more qualitative and limited in scale. A more systematic user study comparing different backends is left for future work.

Third, we rely on existing open-source LLMs and prompt engineering. While the tool-maker paradigm is model-agnostic, actual performance and cost trade-offs may vary across LLM families and deployment settings. Exploring fine-tuned, open-source models for this task is an interesting direction.

Finally, our current evaluation mostly targets technical correctness and visual quality. We do not yet perform large-scale classroom studies measuring actual learning gains. Understanding how different visualization styles and levels of detail affect student learning remains an important open question.

Ethical Considerations

Our system is designed for educational purposes and does not directly handle sensitive user data. However, large-scale generation of educational content could impact existing teaching materials and content creators. We encourage responsible use, including clear attribution when reusing generated materials and avoiding misleading learners with incorrect visualizations.

Our benchmark is constructed from public LeetCode problems and does not include personal or proprietary data. We release our code under the MIT license and dataset to support reproducibility and further research. We do not foresee direct negative societal impacts, but as with any automation tool, misuse (e.g., mass-producing low-quality teaching materials without validation) is possible and should be guarded against. We used AI assistants (e.g., ChatGPT) for language polishing; all content was reviewed and verified by the authors.

References

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2024. Large language models as tool makers. In *International Conference on Learning Representations (ICLR)*. ArXiv:2305.17126.

Eason Chen, Ray Huang, Han-Shin Chen, Yuen-Hsien Tseng, and Liang-Yi Li. 2023a. Gptutor: a chatgpt-powered programming tool for code explanation. *Preprint*, arXiv:2305.01863.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023b. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Preprint*, arXiv:2211.12588.

Yanzhe Chen, Kevin Qinghong Lin, and Mike Zheng Shou. 2025. Code2video: A code-centric paradigm for educational video generation. *Preprint*, arXiv:2510.01174.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, and 181 others. 2025. Deepseek-v3 technical report. *Preprint*, arXiv:2412.19437.

Victor Dibia. 2023. LIDA: A tool for automatic generation of grammar-agnostic visualizations and infographics using large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 113–126, Toronto, Canada. Association for Computational Linguistics.

Yingqiang Ge, Wenyue Hua, Kai Mei, Jianchao Ji, Juntao Tan, Shuyuan Xu, Zelong Li, and Yongfeng Zhang. 2023. Openagi: When llm meets domain experts. *Preprint*, arXiv:2304.04370.

Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Dan Zhang, Diego Rojas, Guanyu Feng, Hanlin Zhao, Hanyu Lai, Hao Yu, Hongning Wang, Jiadai Sun, Jiajie Zhang, Jiale Cheng, Jiayi Gui, Jie Tang, Jing Zhang, and 39 others. 2024. Chatglm: A family of large language models from glm-130b to glm-4 all tools. *Preprint*, arXiv:2406.12793.

Philip Guo. 2013. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE)*.

Steven Halim, Zi Chun Koh, Victor Bo Huai Loh, and Felix Halim. 2012. Learning algorithms with unified and interactive web-based visualization. *Olympiads in Informatics*, 6:53–68.

CHRISTOPHER D. HUNDHAUSEN, SARAH A. DOUGLAS, and JOHN T. STASKO. 2002. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, 13(3):259–290.

Rasha Ahmad Husein, Hala Aburajouh, and Cagatay Catal. 2025. Large language models for code completion: A systematic literature review. *Computer Standards & Interfaces*, 92:103917.

Pooriya Jamie, Reyhaneh HajiHashemi, and Sharareh Alipour. 2025. Utilizing chatgpt in a data structures and algorithms course: A teaching assistant's perspective. In *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, CHI EA '25, page 1–7. ACM.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2025. A survey on large language models for code generation. *ACM Trans. Softw. Eng. Methodol.* Just Accepted.

Max Ku, Thomas Chong, Jonathan Leung, Krish Shah, Alvin Yu, and Wenhu Chen. 2025. Theoremexplainagent: Towards video-based multimodal explanations for llm theorem understanding. *Preprint*, arXiv:2502.19400.

Mingyuan Li, Duan Wang, Erick Purwanto, Thomas Selig, Qing Zhang, and Hai-Ning Liang. 2025. Visualcodemooc: A course platform for algorithms and data structures integrating a conversational agent for enhanced learning through dynamic visualizations. *SoftwareX*, 30:102072.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Preprint*, arXiv:2305.01210.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. 2024. A dynamic llm-powered agent network for task-oriented agent collaboration. *Preprint*, arXiv:2310.02170.

Yaxi Lu, Haolun Li, Xin Cong, Zhong Zhang, Yesai Wu, Yankai Lin, Zhiyuan Liu, Fangming Liu, and Maosong Sun. 2025. Learning to generate structured output with schema reinforcement learning. *Preprint*, arXiv:2502.18878.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. *Preprint*, arXiv:2303.17651.

Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. 2023. Augmented language models: a survey. *Preprint*, arXiv:2302.07842.

mrdoob and contributors. 2025. three.js: Javascript 3d library (release r182). GitHub repository. R182 released on Dec 10, 2025.

Thomas L. Naps, James R. Eagan, and Laura L. Norton. 2000. Jhavé—an environment to actively engage students in web-based algorithm visualizations. In *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education (SIGCSE '00)*, pages 109–113.

Thomas L Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Christopher Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and 1 others. 2002. Exploring the role of visualization and engagement in computer science education. *ACM SIGCSE Bulletin*, 35(2):131–152.

Geliang Ouyang, Jingyao Chen, Zhihe Nie, Yi Gui, Yao Wan, Hongyu Zhang, and Dongping Chen. 2025. nvagent: Automated data visualization from natural language via collaborative agent workflow. *Preprint*, arXiv:2502.05036.

Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-lite: A grammar of interactive graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Preprint*, arXiv:2302.04761.

Clifford A Shaffer, Matthew L Cooper, Alexander J Alon, Monika Akbar, Michael Stewart, Sean Ponce, and Stephen H Edwards. 2010. Algorithm visualization: The state of the field. *ACM Transactions on Computing Education (TOCE)*, 10(3):1–22.

SiliconFlow. 2025. Siliconflow: Lightning-fast ai platform for llms and multimodal models. `https://www.siliconflow.com/`. Accessed: 2025-12-04.

Simon Su, Edward Zhang, Paul Denny, and Nasser Giacaman. 2021. A game-based approach for teaching algorithms and data structures using visualizations. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, page 1128–1134, New York, NY, USA. Association for Computing Machinery.

Till Tantau. 2025. *The TikZ and PGF Packages: Manual for Version 3.1.11a*. Version 3.1.11a (Aug 29, 2025).

The Manim Community Developers. 2024. Manim: A mathematical animation engine.

Euripides Vrachnos and Athanassios Jimoyiannis. 2014. Design and evaluation of a web-based dynamic algorithm visualization environment for novices. *Procedia Computer Science*, 27:229–239. 5th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion, DSAI 2013.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. *Preprint*, arXiv:2203.11171.

Irene Weber. 2024. Large language models as software components: A taxonomy for llm-integrated applications. *Preprint*, arXiv:2406.10300.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. *Preprint*, arXiv:2201.11903.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Xuanle Zhao, Xianzhen Luo, Qi Shi, Chi Chen, Shuo Wang, Zhiyuan Liu, and Maosong Sun. 2025. Chartcoder: Advancing multimodal large language model for chart-to-code generation. *Preprint*, arXiv:2501.06598.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Preprint*, arXiv:2306.05685.

Li Zhong and Zilong Wang. 2024. Can chatgpt replace stackoverflow? a study on robustness and reliability of large language model code generation. *Preprint*, arXiv:2308.10335.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2023. Docprompting: Generating code by retrieving the docs. *Preprint*, arXiv:2207.05987.

Daniel Zingaro, Cynthia Taylor, Leo Porter, Michael Clancy, Cynthia Lee, Soohyun Nam Liao, and Kevin C. Webb. 2018. Identifying student difficulties with basic data structures. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ICER '18, page 169–177, New York, NY, USA. Association for Computing Machinery.

A Qualitative Gallery

We provide a qualitative gallery of generated visualizations across different algorithm families to demonstrate the versatility of our framework beyond a single running example. Figure 6 summarizes cross-method comparisons, while Figures 7–8 provide per-family views of our system's typical visualizations and failure modes used in the case study; Figure 9 further contrasts our method with end-to-end Manim baselines on a challenging dynamic-programming task.

B Open-Source Resources

We plan to release:

- **Code**: the full ALGOGEN system, including VTA-JSON 5.0 specification, validators, renderers, and evaluation scripts.
- **Dataset**: 200 LeetCode tasks in standardized `example/*.txt` format.
- **Evaluation results**: AES / TEA scores and automatic metrics for all methods.
- **Demo videos**: a curated set of high-quality AV videos across algorithm families.
- **Documentation**: a system technical report and user guide.

All resources will be released under the MIT license.

C Experimental Environment

C.1 Python Dependencies

```
python>=3.9
openai>=1.0.0
manim>=0.18.0
```

C.2 System Dependencies

```
# LaTeX (optional, for TikZ rendering)
sudo apt-get install texlive-xetex

# FFmpeg (Manim video encoding)
sudo apt-get install ffmpeg

# Image conversion (optional)
sudo apt-get install pdftocairo
```

We use a SiliconFlow API key to access all LLM models (DeepSeek-V3, Qwen series, GLM-4.6).

D Additional Evaluation Tables

D.1 Renderer Performance

Table 5 reports the runtime characteristics of the three renderers on the 200-task benchmark.

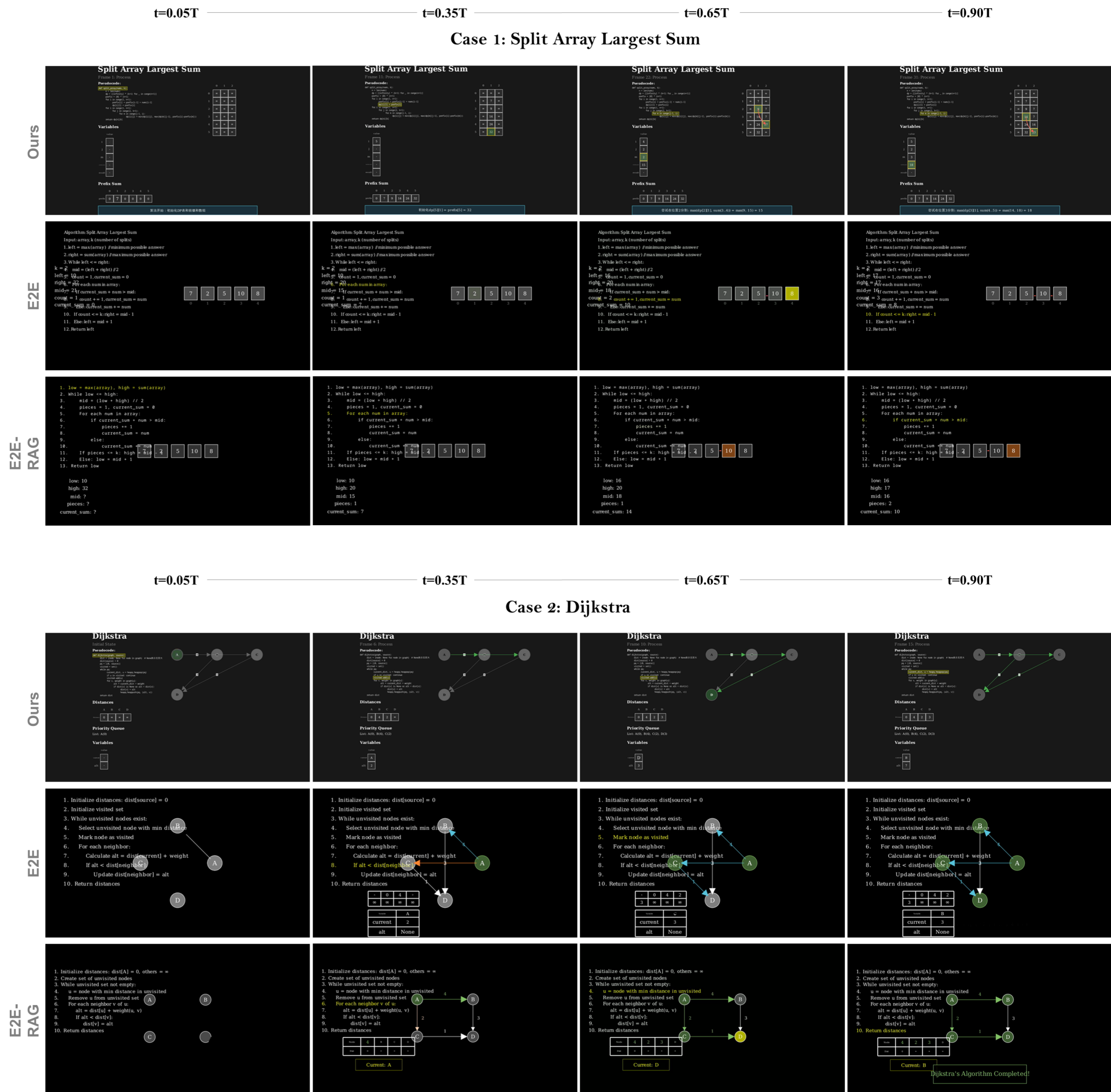

Figure 6: **Gallery of AlgoGen Results and Baselines.** Selected examples from array, graph, dynamic programming, tree, and sorting families, showing representative key frames across methods in a filmstrip layout.

D.2 TEA Evaluation Details

Table 6 reports TEA scores per model under the 0–5 rubric.

E Dataset Construction Details

E.1 Data Collection Pipeline

Our dataset is constructed through a multi-step pipeline (`batch_fetch_all.py`). We first collect 2,530 LeetCode problems via LeetCode's GraphQL endpoint, and then convert them into 3,958 standardized task instances in our system format (each instance corresponds to a concrete input example for a problem):

1. **Batch Fetching**: Retrieve problems from LeetCode's GraphQL endpoint for 6 algorithm families.

2. **Manual Reclassification**: Apply corrections for mislabeled problems (e.g., problems tagged as "Graph" but actually requiring DP).

3. **Format Conversion**: Convert to standardized task specification format, including schema normalization and validation (e.g., graph canonicalization into a nested `graph` object with explicit `directed` edges, and basic type-/field checks). We export two variants of task files (with and without natural-language problem descriptions) to support different evaluation settings.

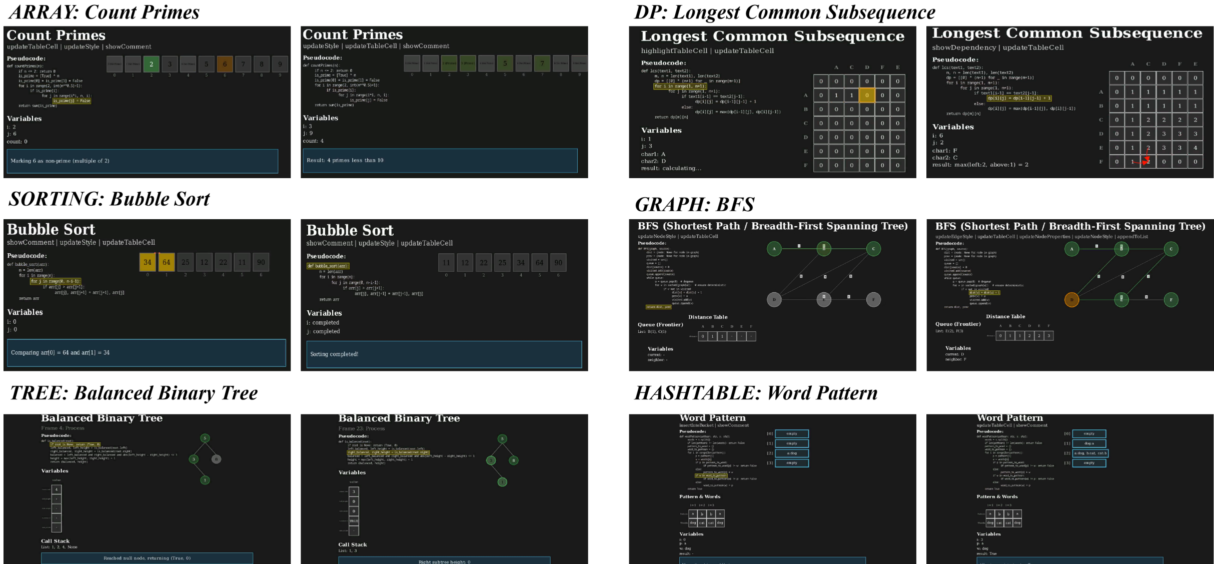

Figure 7: **Case study: representative visualizations from AlgoGen.** Qualitative examples from six algorithm families illustrating that our pipeline can consistently visualize core data structures and step-wise execution without obvious occlusion or overlap. This figure corresponds to the high-quality cases discussed in the case study.

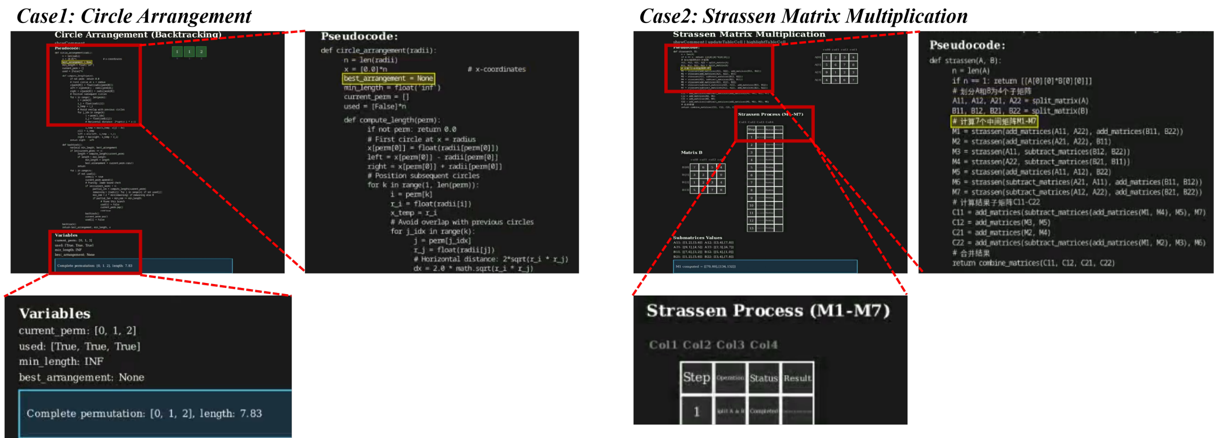

Figure 8: **Case study: failure modes of AlgoGen.** Representative failure cases under high information density, where long pseudocode blocks or large tables cause the canvas to compress visual elements and reduce legibility. This figure corresponds to the limitations discussed in the case study subsection of the main text.

4. **Stratified Sampling**: From 3,958 candidate task instances, sample 200 tasks using `create_small_dataset.py` with:
   - Minimum 15 samples per algorithm family
   - Remaining samples allocated proportionally
   - Random seed = 42 for reproducibility

We collect problems via LeetCode's GraphQL endpoint for research/education only, and our released artifacts are intended for the same purpose. The dataset is a derived research artifact; users should comply with LeetCode's terms of use, and we do not intend it for use outside a research setting. We avoid redistributing any content beyond what is permitted by the original access conditions.

E.2 Task Selection Criteria

1. **Stateful**: Exclude pure mathematical calculations without state changes (e.g., `Power(x, n)`).
2. **Visualizable**: Require data structures with clear geometric representations.
3. **Balanced difficulty**: Easy (35%), Medium (53%), Hard (12%).
4. **Family diversity**: Array (84), DP (29), Sort-

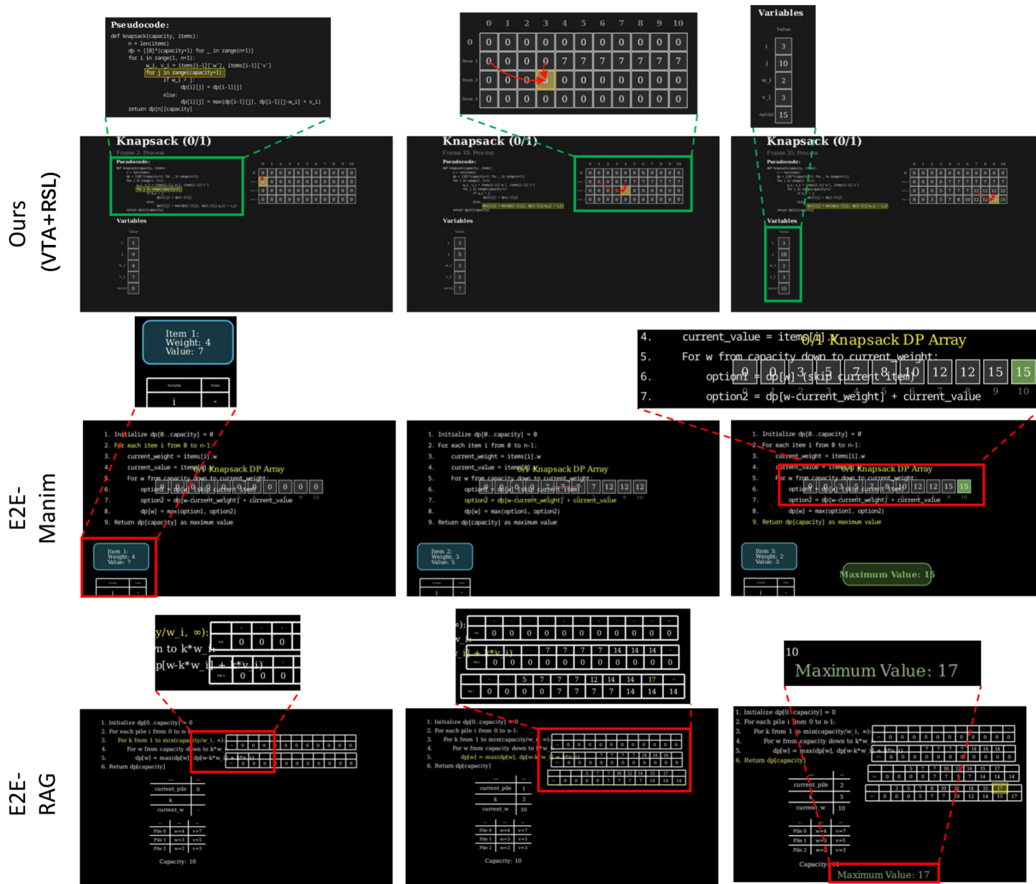

Figure 9: **Qualitative comparison on a complex dynamic-programming task (Knapsack 0/1).** Rows show key frames from **Ours** (Top), **E2E-Manim** (Middle), and **E2E-RAG** (Bottom). **(Top)** Our VTA-based system generates a clean split-screen layout, with precise variable tracking (green boxes) and synchronized code highlighting. **(Middle)** The end-to-end baseline suffers from *layout clutter*, failing to wrap the long DP array. **(Bottom)** The RAG-enhanced model exhibits severe *spatial hallucinations*, causing elements to overlap and rendering the code unreadable (red boxes).

ing (26), Graph (22), Tree (21), Hashtable (18).

E.3 Task File Format

Each task file (`example/*.txt`) contains:

```
Algorithm Snippet (Course Schedule):

- LeetCode Problem ID: 207
- Difficulty: Medium
- Goal: Generate `graph_tracker.py`
- User Request: Create visualization tracker
              for "Course Schedule (Graph)"

- Input:
input_data = {
  "graph": {
    "nodes": [{"id": "A", "label": "A"}, ...],
    "edges": [{"from": "A", "to": "B",
              "weight": 4, "directed": true},
              ↪  ...]
  },
  "source": "A"
}
```

E.4 Quality Control

All tasks are manually reviewed to ensure:

- Valid and representative input data with edge cases.
- Correct algorithm family classification after reclassification.
- Consistent tagging and visual requirements.

F VTA-JSON 5.0 Specification

VTA-JSON 5.0 is the JSON encoding of our Visualization Trace Algebra (VTA) for algorithm visualization, designed to be simple for LLMs to generate via in-context learning while expressive enough to cover common AV patterns.

Model	#Eval	Layout	Attract.	Logic	Acc.&Depth	Consist.	Total
DeepSeek-V3.1	200	17.37	16.72	18.08	**18.44**	18.01	**88.62**
GLM-4.6	200	17.36	**16.75**	18.05	18.37	**18.06**	88.60
Qwen3-235B	199	17.36	16.54	17.90	18.29	18.03	88.13
Average	599	17.36	16.67	18.01	18.37	18.03	88.45

Table 4: AES evaluation results per model (0–20 per dimension, 0–100 total).

Renderer	#Traces	Success	Success rate	Avg. time	Output size
Manim video	200	200	100%	3.03min / task	2.8MB / video
LaTeX/TikZ	200	200	100%	1.04min / task	156KB / frame set
Three.js	200	200	100%	instantaneous	trace.json only

Table 5: Performance of three renderers on the same 200 VTA-JSON traces.

F.1 JSON Structure

A VTA-JSON trace has the following top-level structure:

```
{
  "vta_version": "5.0",
  "algorithm": {
    "name": "Dijkstra Shortest Path",
    "family": "Graph"
  },
  "initial_frame": {
    "data_schema": { ... },
    "data_state": {
      "type": "graph",
      "structure": {
        "nodes": [{"id": "A", "label": "A",
                   "styleKey": "idle",
                 "properties": {"distance": 0}}],
        "edges": [{"from": "A", "to": "B",
                   "weight": 4, "styleKey":
                   ↪  "normal"}]
      }
    },
    "auxiliary_views": [...],
    "styles": { "elementStyles": {...} },
    "pseudocode": ["1. Initialize distances",
    ↪  ...]
  },
  "deltas": [
    {
      "action_description": "Select node A",
      "code_highlight": 2,
      "operations": [[
        {"op": "updateNodeStyle",
         "params": {"ids": ["A"], "styleKey":
         ↪  "current"}}
      ]]
    }
  ],
  "required_extensions":
  ↪  ["vta-ext-primitive-graph"]
}
```

F.2 Supported Operations

VTA defines approximately 30 atomic operations organized by view type, which are serialized in VTA-JSON traces as op codes:

F.3 Delta Semantics

VTA-JSON uses *deltas* rather than absolute frames. Each delta describes how to transform the previous frame into the next one, making it natural to map onto animations (e.g., smooth movement, fade-in/fade-out) and to align trace steps with algorithm operations. The `operations` field is a 2D array, where the inner arrays group logically simultaneous operations.

F.4 Schema Validation

A validator enforces VTA-JSON 5.0 invariants:

- `vta_version` must be string `"5.0"` (not numeric)
- `operations` must be 2D arrays (`[[...]]`)
- No `Infinity` values (use `null` for undefined)
- Graph edge endpoints must reference existing nodes
- `code_highlight` must be integer or integer array

G RSL Specification

RSL (Rendering Style Language) is a declarative DSL that controls *how* VTA-JSON traces are rendered. It is implemented in `manim/test_v2/` with three components: `rsl_schema.json` (JSON Schema), `rsl_generator.py` (LLM-based generation), and `rsl_interpreter.py` (conversion to render config).

G.1 Schema Structure

An RSL configuration contains five top-level fields:

Model	#Eval	Layout	Attract.	Logic	Acc.&Depth	Consist.	Total
DeepSeek-V3.1	200	2.92	1.12	1.75	**4.99**	**4.93**	**4.70**
GLM-4.6	200	2.81	1.09	1.56	**4.99**	**4.93**	4.66
Qwen3-235B	199	2.67	1.04	1.20	**4.99**	**4.94**	4.62
Average	599	2.79	1.09	1.51	4.99	4.93	4.65

Table 6: TEA evaluation results (0–5 scale) for three models.

View Type	Core Operations
Array	`updateStyle`, `moveElements`, `shiftElements`, `updateValues`
Graph	`updateNodeStyle`, `updateNodeProperties`, `updateEdgeStyle`, `addNode`, `removeNode`
Tree	`addChild`, `reparent`, `rotate`
Hashtable	`insertIntoBucket`, `rehash`, `highlightCollision`
Table (DP)	`updateTableCell`, `highlightTableCell`, `showDependency`
Generic	`showComment`, `hideComment`, `appendToList`, `popFromList`

Table 7: VTA/VTA-JSON 5.0 operation categories by view type.

```
{"meta": {"rsl_version": "0.1"},
 "theme": {"background": "#1A1A1A",
   "text": "#FFFFFF", "primary": "#3498DB"},
 "timeline": {"transition": 0.5, "pause": 0.3},
 "layout": {"main": {
   "type": "force_directed",  // or grid/matrix
   "params": {"node_spacing": 2.0}}},
 "rules": [{"when": {"op": "updateNodeStyle"},
   "do": {"animation": {"variant": "pulse"}}}]}
```

G.2 Layout Types

Supported layout types: `force_directed`, `hierarchical`, `circular`, `grid`, `matrix`, `horizontal_array`. Layout parameters include `node_spacing` (1.0–10.0), `edge_curve` (-1.0–1.0), `cell_size` (0.3–2.0).

G.3 LLM-Driven Generation

The `rsl_generator.py` extracts trace features (algorithm family, data type, scale, operations used) and prompts an LLM to generate an RSL config. The output is validated against the JSON Schema and semantic checks (e.g., only allowed VTA operation names in `rules[].when.op`).

G.4 Safety Constraints

- Numeric params bounded: `transition` $\in [0.1, 2.0]$, `pause` $\in [0, 1.0]$
- Layout types constrained to predefined enums
- Animation variants: `pulse`, `glow`, `shake`, `fade`, `morph`
- Invalid configs fall back to defaults

H Formal Properties of VTA

H.1 Monoid Structure of Primitive Operations

Theorem A.1 (VTA monoid of primitive operations). Let Op be the finite set of primitive visual operation symbols defined in our VTA-JSON 5.0 specification (Appendix F; e.g., `updateStyle`, `updateNodeStyle`, `updateTableCell`). Let $M := Op^*$ be the set of all finite sequences (including the empty sequence) over Op. Define a binary operation $* : M \times M \to M$ by sequence concatenation: for $a, b \in M$, $a * b$ is the sequence obtained by appending b to a. Let $\epsilon \in M$ denote the empty sequence. Then $(M, *, \epsilon)$ is a monoid.

Proof. We first fix notation. Each element of Op denotes a primitive visual operation such as updating a table cell or changing a node style. An element of $M = Op^*$ is therefore a finite sequence $w = (o_1, \dots, o_n)$ of such primitive operations; the empty sequence is denoted by ϵ.

Closure. By construction, every element of M is a finite sequence of symbols from Op. If $a, b \in M$, then $a * b$ is obtained by concatenating two finite sequences over Op. The result is again a finite sequence over Op, so $a * b \in M$. Therefore the binary operation $* : M \times M \to M$ is closed on M.

Associativity. Concatenation of finite sequences is associative: for any $a, b, c \in M$, first concatenating a and b and then concatenating the result with c yields exactly the same sequence as first concatenating b and c and then concatenating the result with a. Formally,

$$(a * b) * c = a * (b * c) \qquad \text{for all } a, b, c \in M.$$

This is the standard associativity property of free monoids over a generating set Op.

Identity element. Let $\epsilon \in M$ be the empty sequence. For any $a \in M$, concatenating ϵ on the left or on the right leaves a unchanged:

$$\epsilon * a = a, \qquad a * \epsilon = a.$$

Thus ϵ is a two-sided identity element for $(M, *)$.

Combining closure, associativity, and the existence of an identity element, we conclude that $(M, *, \epsilon)$ is a monoid. □

Remark. In the main text, we equip the visual state space S with a right action of this monoid. Each primitive operation $o \in Op$ is interpreted as a (partial) state transformer $[\![o]\!] : S \rightharpoonup S$, and a sequence $w = (o_1, \ldots, o_n) \in M$ acts on a state $s \in S$ via the composite $[\![o_n]\!] \circ \cdots \circ [\![o_1]\!](s)$. This satisfies the usual action law $s \cdot (a * b) = (s \cdot a) \cdot b$, so VTA can be viewed as a typed visual state space equipped with a monoid action of primitive operations.

I Prompt Engineering

Our prompts are composed of modular components. Due to space limits, we present the structural skeleton and key instructions below. Full prompts are available in the code repository.

I.1 Tracker Generation Prompt

The tracker generation prompt (`vta_unified_v2.txt`, ∼43KB) contains six modules:

Module	Purpose
Core Philosophy	Three-version self-verification process
Algorithm First	Correctness as highest priority
13 Hard Rules	Non-negotiable VTA-JSON constraints
Code Structure	Required ordering of code sections
Family Guidelines	Graph/DP/Sorting specific rules
VTA-JSON 5.0 Spec	Complete operation definitions

Table 8: Tracker prompt module overview.

System Prompt: Tracker Generation ($\mathcal{P}_{\text{tracker}}$)

Role: You are an algorithm visualization expert generating Python trackers that emit VTA-JSON 5.0 traces.
Three-Version Process: 1. **Version 1 (Draft)**: write an initial tracker focusing on algorithm logic.
2. **Self-Check**: verify all hard rules, VTA-JSON field naming, and input/output alignment.
3. **Version 2 (Corrected)**: fix every issue found in self-check.
4. **Final Verification**: mentally simulate the first few iterations and check that data-structure updates and code highlights are correct.
5. **Version 3 (Final Submission)**: the only version we keep and execute.
Hard Constraints: We define a set of non-negotiable constraints covering VTA-JSON structural invariants (version strings, array dimensions, field naming), data type restrictions, and code quality requirements. Representative examples include ensuring `operations` is always a 2D array and that render operations never directly modify Python state. The complete constraint set is available in our codebase.
Algorithm-Family Guidelines: We provide algorithm-family-specific guidelines for graph algorithms (e.g., frontier management, deterministic traversal order), dynamic programming (e.g., table-based state representation), and sorting algorithms (e.g., element comparison and swap visualization). These guidelines ensure consistent and pedagogically sound visualizations across algorithm families.
VTA-JSON 5.0 Specification: *[The full JSON schema and ∼30 operation definitions are provided in the repository under* `vta_specification/` *and omitted here for brevity.]*

I.2 Generated Tracker Example

Below is a simplified excerpt from a generated tracker for "Count Primes" (Sieve of Eratosthenes), illustrating the structure of LLM-generated code and how a tracker emits a VTA-JSON trace:

```
import json

input_data = {"array": [1, 2, 3, 4, 5]}

def main():
    n = len(input_data["array"])
    is_prime = [True] * (n + 1)
    is_prime[0] = is_prime[1] = False

    trace = {
        "vta_version": "5.0",
        "algorithm": {"name": "Count Primes",
                      "family": "Sieve of
                      ↪  Eratosthenes"},
        "required_extensions":
        ↪  ["vta-ext-primitive-array"],
        "initial_frame": {
            "data_state": {"type": "array",
                "structure": [{"index": i,
                ↪  "value": v,
                   "state": "idle"}
                  for i, v in enumerate(input_⌋
                  ↪  data["array"])]},
            "pseudocode": ["1. Initialize sieve",
            ↪  ...],
            "styles": {"elementStyles": {...}}
        },
        "deltas": []
    }

    # Algorithm execution with VTA-JSON
    ↪  operations
    for i in range(2, int(n**0.5) + 1):
        if is_prime[i]:
            # Emit VTA-JSON operation before
            ↪  state change
```

```
            trace["deltas"].append({
                "code_highlight": 5,
                "operations": [[{"op":
                ↪  "updateStyle",
                    "params": {"indices": [i],
                               "styleKey":
                               ↪  "current"}}]]
            })
            for j in range(i*i, n+1, i):
                is_prime[j] = False  # Python
                ↪  state first
                # Then emit render operation
                trace["deltas"].append({...})

    with open("trace.json", "w") as f:
        json.dump(trace, f)

if __name__ == "__main__":
    main()
```

The key pattern is: **update Python state first, then emit VTA-JSON render operations**. This ensures the trace accurately reflects algorithm execution while keeping the tracker code readable and teachable.

I.3 RSL Generation Prompt ($\mathcal{P}_{\text{rsl}}$)

System Prompt: RSL Generation ($\mathcal{P}_{\text{rsl}}$)

Role: You are a rendering-style expert. Output only valid RSL JSON.
VTA-JSON Context (Read-Only): Algorithm name and family, data type and scale, number of frames, and the list of VTA-JSON operations used are summarized and passed as context; the trace itself must not be modified.
Design Goals: Improve layout clarity, visual contrast, and animation rhythm while respecting the trace semantics. In particular, `rules[].when.op` must use actual VTA operation names (e.g., `updateNodeStyle`, `updateTableCell`), not semantic labels like "visit_node".
Schema Snippet: The prompt inlines the RSL JSON Schema (meta/theme/timeline/layout/rules/annotations) so that the model can validate field names and ranges.
[We omit the full schema here; see `manim/test_v2/rsl_schema.json` *for details.]*

I.4 Error-Guided Repair Prompt ($\mathcal{P}_{\text{repair}}$)

System Prompt: Error-Guided Repair ($\mathcal{P}_{\text{repair}}$)

Context: If executing the tracker or validating the VTA-JSON trace fails, we summarize the error and feed it back to the model.
Error Block:

```
[Previous Error]
{error_type}: {error_message}
Location: line {line_number}, variable
↪  {var_name}
```

Repair Instructions (Abstracted): 1. Analyze the error type, location, and offending variable.
2. Apply targeted fixes guided by the summarized error (e.g., resolving schema violations or type mismatches) while preserving the intended algorithmic behavior.
3. Keep unrelated code intact and focus edits on the minimal changes needed to pass validation.
4. Return a complete, self-contained Python file; do not output patches.

I.5 Algorithm Correctness Evaluator ($\mathcal{P}_{\text{tier2}}$)

System Prompt: Algorithm Correctness Evaluator ($\mathcal{P}_{\text{tier2}}$)

Role: You are an algorithm expert reviewing a Python tracker for a single visualization example.
Inputs: Algorithm name and family, an optional reference pseudocode snippet (if available), and the full `tracker.py` source including VTA-JSON-specific trace-generation logic.
Rubric (100 points total):

- 50 pts *Algorithm logic correctness*: coverage of key steps and execution order for the given example.
- 30 pts *VTA-JSON compliance*: required fields, operation names, data types, and extension declarations.
- 10 pts *Result presentation*: how final outputs and important variables are surfaced in the trace.
- 10 pts *Code quality*: naming, comments on key steps, and overall structure of the trace-generation code.

Evaluation principles: Focus on whether the tracker correctly implements the intended algorithm for the visualization input rather than speculating about unseen corner cases; distinguish semantic errors (wrong results, missing steps) from stylistic differences (e.g., using <= vs. <); when uncertain about a critical branch, err on the conservative side and record the source of uncertainty in the issue list instead of assigning a near-perfect score.
Output format: The model must return a strict JSON object with per-dimension scores, a 0–50 `algorithm_logic` subscore (later normalized to the reported 0–100 "algorithm-correctness" metric), a list of issues (`severity/category/description`), a list of strengths, and a short overall assessment.

We apply $\mathcal{P}_{\text{tier2}}$ to all 200 LeetCode tasks for each tracker generator (DeepSeek-V3.1, Qwen3-235B, GLM-4.6). The static evaluator itself is always DeepSeek-V3.1, acting purely as a code reviewer. On the DeepSeek-generated trackers, this setup consumes on average roughly 5.1k prompt tokens and 0.3k completion tokens per tracker (about 5.4k tokens in total), providing a scalable yet fine-grained view of code-level correctness.

Model	Alg. logic	VTA compl.	Return disp.	Code qual.
DeepSeek-V3.1	49.9	28.5	9.9	8.8
Qwen3-235B	49.4	28.0	8.9	8.7
GLM-4.6	49.5	28.5	9.8	8.8

Table 9: Average tier-2 rubric scores (0–50 for algorithm logic, 0–30 for VTA-JSON compliance, 0–10 for return presentation and code quality) over 200 tasks per generator model.

Per-family end-to-end success. Table 10 reports end-to-end success rates by algorithm family and the absolute improvement of our VTA+RSL pipeline over the `manim_direct` baseline.

J Failure Analysis of End-to-End Baselines

J.1 Breakdown of Failures

Analysis of 35 failed cases from `manim_direct`:

J.2 Representative Failure Cases

Case 1: API Parameter Error.

```
# Generated (incorrect)
self.pseudocode[line].set_background_stroke(BLA
↪  CK, 3)

# Correct usage
self.pseudocode[line].set_background_stroke(
    color=BLACK, width=3)
```

The LLM used positional arguments, but Manim v0.18 requires keyword arguments. Despite correct documentation in the prompt, the model relied on pre-training patterns from older Manim versions.

Case 2: Non-existent Attribute.

```
# Generated (incorrect)
num_nodes = self.vars_table.shape[0]

# Error: 'Table' has no attribute 'shape'
# Correct: len(self.vars_table.get_rows())
```

The LLM incorrectly transferred `numpy` array patterns to Manim `Table` objects—a "negative transfer" from similar libraries.

Case 3: Rendering Timeout. For DP algorithms, the LLM generated fine-grained animations for each cell update (1000+ `FadeIn` calls), exceeding 20-minute timeout. The model cannot estimate animation computational cost.

J.3 Why RAG Does Not Help

RAG with 12K-token Manim documentation *reduces* success rate from 82.5% to 79.0%. Analysis of 42 RAG failures:

- 60% are algorithm logic errors (RAG provides API knowledge, not algorithmic reasoning)
- 40% are API knowledge issues (even with correct docs, LLM relies on pre-training memory)
- Information overload: 12K tokens dilute attention on critical constraints

This demonstrates that the bottleneck is *task architecture*, not knowledge availability. Our VTA/VTA-JSON pipeline achieves near-perfect success by simplifying the task structure rather than merely augmenting knowledge.

K Baseline Implementation Details

To ensure a fair comparison, we design comprehensive system prompts for the end-to-end baselines: `manim_direct_novta`, `manim_direct`, and `manim_direct_rag`. Despite detailed API guidelines and layout instructions, these baselines still struggle with logical consistency and spatial layout, as summarized in Table 2.

K.1 Prompt for `manim_direct_novta`

This prompt allows the model to freely design Manim visualizations while enforcing basic correctness and usability constraints, without relying on our structured VTA/VTA-JSON IR.

System Prompt: Free Manim Generation ($\mathcal{P}_{\text{free}}$)

Role: You are a Manim author with full creative freedom to design the "best" visualization for the given algorithm.
Core Principle: Algorithmic correctness is the top priority.
Strict Prohibitions: 1. Do not ask the user any questions or request more information. 2. Do not output partial code or natural-language explanations.
Hard Rules (Abstracted): 1. Code must run with `manim -pqh scene.py AlgorithmScene`. 2. Class name must be `AlgorithmScene(Scene)` with a complete `construct()`. 3. Pseudocode must be shown somewhere on screen and cover key steps. 4. No Chinese variable names. 5. Graph algorithms must sort adjacency lists to ensure determinism. 6. Use `VGroup` to manage objects; remove temporary objects to avoid accumulation. 7. Animation run time in each step should be within $[0.3, 0.8]$ seconds. 8. Maintain readable text and sufficient color contrast; elements must stay within the canvas.
Manim Cheat Sheet (Provided in Context): *[Dozens of examples for text, geometric primitives, tables, layout helpers (`next_to`, `arrange`), and common animations (`FadeIn`, `Create`, `.animate.set_color`, etc.) are provided here. We omit ∼100+ lines of API examples for brevity.]*

	Sort	Array	DP	Tree	Graph	Hash	Avg.
Ours (VTA+RSL)	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Manim_direct	88.5	89.3	58.6	95.2	72.7	77.8	80.4
Gap (↑)	+11.5	+10.7	+41.4	+4.8	+27.3	+22.2	+19.6

Table 10: End-to-end success rate (%) by algorithm family on ALGOGEN-Bench. **Gap** reports absolute improvements of Ours over Manim_direct.

Error Type	**Count**	**Percentage**
API parameter errors	18	51.4%
Non-existent attributes	7	20.0%
Class confusion	5	14.3%
Rendering timeout	5	14.3%

Table 11: Distribution of failure types in end-to-end Manim generation.

Recommended Code Skeleton:

```
from manim import *
import numpy as np

input_data = {...}

class AlgorithmScene(Scene):
    def construct(self):
        self.show_title()
        self.create_pseudocode()
        self.create_visualization()
        self.run_algorithm()
        self.show_result()

    # Helper methods are free-form but
    ↪  encouraged.
```

1111

K.2 Prompt for `manim_direct`

1112 1113 1114 1115 1116

The `manim_direct` baseline uses a more structured prompt that mirrors the three-version self-verification idea, but still generates Manim Scenes directly without using VTA/VTA-JSON.

System Prompt: Structured End-to-End Generation ($\mathcal{P}_{\text{direct}}$)

Role: You are an expert Manim developer. Write a complete Python script using Manim Community Edition to visualize the given algorithm.
Three-Version Process: 1. **Version 1 (Draft)**: Produce an initial Manim Scene. 2. **Self-Check**: Inspect algorithm logic, layout, and API usage; list problems explicitly. 3. **Version 2 (Corrected)**: Fix all issues found in self-check. 4. **Final Verification**: Re-check algorithm and Manim APIs. 5. **Version 3 (Final Submission)**: Polished code that will be executed.
Key Constraints (Abstracted):

- Code must run with `manim -pql scene.py AlgorithmScene`.
- Include a detailed pseudocode panel; highlighting must stay synchronized with algorithm steps.
- Sort graph adjacency lists for reproducible traversal order.
- Use `VGroup`, `arrange`, and `next_to` to avoid overlaps and keep all elements within the frame.
- Use reasonable animation timing (0.3–0.8s), readable fonts, and high-contrast colors.

1117

Layout Specification (Right-vs-Left Panels): - Left panel: stacked pseudocode, variable table, distance table, and frontier queue. - Right panel: main data view (array visualization, graph, or DP table).
Code Skeleton:

```
from manim import *

class AlgorithmScene(Scene):
    def construct(self):
        self.setup_layout()
        self.create_pseudocode()
        self.create_data_view()
        self.create_auxiliary_views()
        self.run_algorithm()
        self.show_result()
```

1118

K.3 Prompt for `manim_direct_rag`

1119

The RAG-enhanced baseline uses the same core prompt as `manim_direct`, but prepends retrieved Manim API documentation.

1120 1121 1122

System Prompt: RAG-Enhanced Generation

Role: You are an expert Manim developer.
Retrieved Context (Top-K API Docs): *[Before the main instruction, we insert a large block of Manim API snippets retrieved from* `exp/manim_direct_rag/manim_api_knowledge.json`*, including class definitions (e.g.,* `Graph`*,* `Table`*), typical method usages, and layout/animation examples. The total size of this context is roughly 12K tokens; we omit the ∼3000-word documentation here for brevity.]*
Instruction (Simplified): Using only the documentation and guidelines above, write a complete Manim script that visualizes the given algorithm description, following the same layout and correctness constraints as `manim_direct`.

1123

1124 # L Statements

1125 ## L.1 Conflict of Interest

1126 The authors declare no conflict of interest.

1127 ## L.2 Data and Code Availability

1128 All code, datasets, and the VTA-JSON specifica-
1129 tion used in this paper will be made publicly avail-
1130 able on GitHub under the MIT license upon publi-
1131 cation.