ManiBench

A Benchmark for Testing Visual-Logic Drift and Syntactic Hallucinations in Manim Code Generation

Oli, Nabin

February 20, 2026

Abstract

Traditional code-generation benchmarks like HumanEval and MBPP excel at testing logic and syntax, but they fall short when code must translate into dynamic, pedagogical visuals. We introduce ManiBench, a specialized benchmark designed to evaluate LLM performance in generating Manim CE (Community Edition) code—a domain where temporal fidelity and version-aware API correctness are paramount. ManiBench addresses two critical failure modes prevalent in LLM outputs: *Syntactic Hallucinations* (generating code that is grammatically valid Python but references non-existent Manim functions, outdated or deprecated APIs, undefined classes, or calls that break under specific library versions) and *Visual-Logic Drift* (occurrences where generated visuals diverge from intended mathematical logic, such as missing events, incorrect causal relationships, timing errors, or the model struggling to animate a concept). The benchmark aims to collect 150–200 problems, launching with a pilot of 12 high-quality challenges across five difficulty levels. These span domains including calculus, linear algebra, probability, topology, and AI. Task types are uniquely structured into categories such as drift-sensitive transformations, debugging, version-conflict traps, and multi-scene narratives. Each problem is backed by reference code analysis of the original 3Blue1Brown ManimGL source (∼53,000 lines total, 143 scene classes, 145 documented GL→CE incompatibilities). To move beyond simple test-case-based checks, ManiBench employs a four-tier scoring framework: (1) *Executability* (Pass@1): the fraction of outputs running without exceptions or deprecated imports; (2) *Version-Conflict Error Rate*: the frequency of runs triggering mixed-API or legacy errors; (3) *Alignment Score*: the weighted fraction of required visual events that are both present and temporally accurate; and (4) *Coverage Score*: a four-dimensional measure of pedagogical element density spanning mathematical annotations, visual mapping, numeric evidence, and structural clarity. An accompanying open-source evaluation framework automates metric computation across six LLMs and five prompting strategies. By formalizing the requirements for temporal and syntactic precision, ManiBench provides a foundational testbed for the next generation of automated educational content and visual-logic synthesis.

Keywords: Syntactic Hallucinations, Visual-Logic Drift, Manim CE, Code Generation, Benchmarking

Contents

1 Introduction

The rise of large language models (LLMs) has dramatically accelerated code-generation research. Benchmarks like HumanEval, MBPP, and APPS have become standard evaluation tools for assessing LLM coding ability. However, these benchmarks primarily focus on three criteria:

- **Logic correctness**: Does the code solve the algorithmic problem?
- **Syntax validity**: Does the code parse and execute without errors?
- **Output matching**: Do computed results match expected values?

These criteria are insufficient for domains where code generates continuous, time-dependent visual outputs. Manim, a Python animation engine created by Grant Sanderson (3Blue1Brown), generates mathematical animations by composing scene objects, applying transformations, and controlling timing. A Manim script can be *syntactically valid* yet produce:

- **Incorrect visual semantics**: an animation that moves in the wrong direction;
- **Timing misalignments**: events that occur out of order or at wrong times;
- **Pedagogical failure**: an animation that obscures rather than clarifies the concept.

Additionally, Manim exists in two major versions:

- **Manim CE (Community Edition)**: open-source, actively maintained, with a modern API;
- **Manim GL (3B1B's version)**: the original version, using some deprecated constructs, hand-optimized for performance.

LLMs frequently mix APIs from both versions or reference functions that have been moved or renamed, producing code that fails under specific library versions.

1.1 Contributions

ManiBench makes four key contributions:

1. **Formalized Visual-Logic Metrics.** We define an *Alignment Score* and a four-dimensional *Coverage Score* to capture whether generated animations match pedagogical intent, beyond mere syntactic validity.
2. **Version-Aware Evaluation.** We explicitly test version-conflict errors and deprecated API usage, with 145 documented GL$\rightarrow$CE incompatibilities across eight categories, measuring whether code adheres to a specific Manim version's API contract.
3. **Curated Pilot Dataset with Reference Code Analysis.** We provide 12 hand-crafted benchmark problems drawn from 3Blue1Brown's published videos, backed by comprehensive analysis of $\sim$53,000 lines of original source code including 143 scene classes, 120 visual techniques, and detailed visual-event specifications.
4. **Automated Evaluation Framework.** We release an open-source evaluation pipeline supporting six LLMs across five prompting strategies, with automated metric computation for executability, version conflicts, alignment, and coverage.

2 Problem Definition

2.1 The Two Failure Modes

2.1.1 Syntactic Hallucinations

An LLM generates code that:

- references non-existent classes (e.g., `VMobject` with incorrect spelling);
- uses deprecated functions (e.g., `mobject.scale()` instead of `mobject.scale_to_fit_width()`);
- calls methods with incorrect signatures;
- mixes Manim GL syntax with Manim CE (e.g., using OpenGL-specific rendering calls in CE).

Example:

```python
# HALLUCINATED: class does not exist
circle = MCircle(color=BLUE)  # Should be Circle

# HALLUCINATED: deprecated method
circle.apply_matrix([[1, 0], [0, 1]])
# CE removed in favor of apply_complex_function
```

Listing 1: Syntactic hallucination examples.

2.1.2 Visual-Logic Drift

An LLM generates code that:

- omits required visual events (e.g., gradient descent step without showing dot movement);
- implements events in the wrong order (e.g., loss curve updates before parameter updates);
- uses incorrect timing (animations too fast, pauses missing);
- fails to show causal relationships (e.g., showing a result without showing the derivation).

Example:

```python
# DRIFTED: Gradient descent without showing step updates
def construct(self):
    # Shows loss curve but dot doesn't move downhill
    loss_curve.animate.points = new_points
    # Missing: dot.animate.move_to(new_point)
```

Listing 2: Visual-logic drift example.

2.2 Evaluation Challenges

Subjectivity of "correct." What counts as a correct gradient descent animation? Must the dot move along the loss surface? Must the curve update dynamically? Must the learning rate shrink?

Version fragmentation. A script that passes in Manim CE may fail in Manim GL. We must specify which version(s) the code targets.

Temporal semantics. Unlike static code output (e.g., classification accuracy), animations have temporal semantics. An event can be present but timed incorrectly, creating pedagogical failure.

3 Benchmark Design

3.1 Metric Definitions

3.1.1 Metric 1: Executability (Pass@1)

Definition 3.1 (Executability). *The fraction of generated outputs that run without raising exceptions or using deprecated imports:*

$$\textit{Executability} = \frac{\textit{number of successful executions}}{\textit{number of total attempts}}. \tag{1}$$

Success criteria:
- Script completes without runtime exception.
- No deprecated imports detected (scanned via regex or AST analysis).
- No warnings from Manim's deprecation system.

Failure cases:
- Import error (e.g., `from manim import NonExistent`).
- Runtime `AttributeError` (e.g., `mobject.invalid_method()`).
- Type error (e.g., passing the wrong type to a function).
- Unhandled exception during scene rendering.

3.1.2 Metric 2: Version-Conflict Error Rate

Definition 3.2 (Version-Conflict Error Rate). *The frequency with which generated code triggers errors specific to version constraints:*

$$\textit{VCER} = \frac{\textit{number of mixed-API or legacy errors}}{\textit{number of total attempts}}. \tag{2}$$

Tracked errors:
- GL-specific syntax in CE code.
- CE-only syntax in GL code.
- Calls to renamed or moved functions.
- Signature mismatches due to API evolution.

3.1.3 Metric 3: Alignment Score

Definition 3.3 (Alignment Score). *The weighted fraction of required visual events that are both present and temporally accurate:*

$$\textit{Alignment} = \frac{\sum_i w_i \cdot p_i \cdot t_i}{\sum_i w_i}, \tag{3}$$

where w_i is the importance weight of event i ($0 \le w_i \le 1$), $p_i = 1$ if event i is present (0 otherwise), and $t_i = 1$ if event i occurs at the expected time (0 otherwise).

3.1.4 Metric 4: Coverage Score

Definition 3.4 (Coverage Score)**.** *The density of pedagogical elements, computed as a weighted sum over four sub-dimensions:*

$$Coverage = \sum_{d \in \mathcal{D}} \alpha_d \cdot \frac{|elements\ present\ in\ d|}{|elements\ expected\ in\ d|}, \tag{4}$$

where $\mathcal{D} = \{Math,\ Visual,\ Numeric,\ Structural\}$ and α_d are dimension weights.

The four sub-dimensions and their weights are:

1. **Mathematical Annotation** ($\alpha = 0.35$): formulas, `Tex`/`MathTex` objects, textual labels, variable annotations, and LaTeX commands.
2. **Visual Mapping** ($\alpha = 0.30$): consistent color coding (`set_color`, `set_fill`), arrow indicators, dot markers, surrounding rectangles, and gradient coloring.
3. **Numeric Evidence** ($\alpha = 0.20$): `DecimalNumber`, `Integer`, `ValueTracker`, `NumberLine`, `Axes`, plotted functions, and displayed computed values.
4. **Structural Clarity** ($\alpha = 0.15$): `VGroup`/`Group` organization, `arrange()` layouts, paced `wait()` pauses, `LaggedStart`/`Succession` sequencing, and method decomposition.

3.2 Task Categories

ManiBench organizes problems into five categories:

1. **Direct Visualization (40%).** Prompt → Python code (classic code generation). Difficulty levels 1–3. Metric focus: Executability, Alignment Score.
2. **Drift-Sensitive (20%).** Given a script and a required temporal transformation, detect whether the visual output matches intent. Difficulty levels 2–4. Metric focus: Alignment Score, Coverage Score.
3. **Debugging (20%).** Broken code → fix (repair task). Difficulty levels 2–4. Metric focus: Executability, Alignment Score.
4. **Version-Conflict Traps (10%).** Code with tempting outdated syntax; evaluate whether the model recognizes version constraints. Difficulty levels 3–5. Metric focus: VCER, Executability.
5. **Multi-Scene Narrative (10%).** Hardest tier: multi-scene scripts combining multiple domains. Difficulty levels 4–5. Metric focus: all metrics.

3.3 Difficulty Levels

Level 1 (Trivial). Animate simple objects (circles, squares, text). Expected executability: $>95\%$.

Level 2 (Basic). Animate a transformation or a simple mathematical concept. Expected executability: 80–90%.

Level 3 (Intermediate). Combine multiple transformations; show a mathematical relationship. Expected executability: 70–80%.

Level 4 (Advanced). Multi-step derivation, temporal synchronization, pedagogical clarity. Expected executability: 50–70%.

Level 5 (Expert). Complex concept, multiple scenes, advanced Manim features. Expected executability: 30–50%.

4 Benchmark Dataset

4.1 Pilot Dataset: 12 Problems

The pilot dataset includes 12 hand-curated problems drawn from 3Blue1Brown's published videos. Each problem includes:

1. a natural-language problem statement,
2. video source (YouTube link and timestamp),
3. required visual events with formal specifications,
4. difficulty level (1–5),
5. task category,
6. success criteria for Executability, Alignment, and Coverage, and
7. reference implementation notes (not shared with models; for human evaluation only).

4.1.1 Problem 1: Colliding Blocks Compute π

Metadata: Video ID: `6dTyOl1fmDo`; Category: Drift-Sensitive, Multi-Scene; Difficulty: 4; Domain: Physics, Numerics.

Problem Statement. Write Manim code to animate the collision of two blocks sliding on a frictionless surface. Block A (mass M) starts at rest. Block B (mass m) approaches from the left with velocity v_0. After elastic collision, count the total number of collisions. If $m/M = 0.01$, exactly π wall collisions occur. The animation must show: (1) Block A at $x = 10$, Block B at $x = 0$ moving right; (2) velocity vectors above each block; (3) a collision counter incrementing at each collision; (4) velocity updates after each collision (calculated via elastic collision formulas); (5) the final state with Block B at rest and Block A moving away; and (6) text displaying the collision count.

Required Visual Events (weights in parentheses): blocks move and collide (0.9); collision counter increments correctly (0.8); velocity vectors update after collision (0.7); final text displays collision count (0.6).

Success Criteria: Executability ≥ 0.70; Alignment ≥ 0.70; Coverage ≥ 0.75.

4.1.2 Problem 2: Gradient Descent—How Neural Networks Learn

Metadata: Video ID: `IHZwWFHWa-w`; Category: Direct Visualization, Drift-Sensitive; Difficulty: 3; Domain: Machine Learning, Calculus.

Problem Statement. Create a Manim scene animating gradient descent on a 2D loss landscape. Show: (1) a parametric surface $z = L(w_1, w_2)$; (2) a dot starting at a high-loss location; (3) at each step, compute ∇L, move the dot in the direction of $-\nabla L$, and update a loss curve; (4) 5–10 steps of descent with diminishing step size; (5) arrows indicating gradient direction; and (6) axis labels w_1, w_2, and "Loss."

Required Visual Events (weights): surface visualized (0.8); dot at initial location (0.8); gradient arrow shown and updated (0.7); dot moves downhill (0.9); loss curve plots historical values (0.8); step size diminishes (0.6).

Success Criteria: Executability ≥ 0.95; Alignment ≥ 0.75; Coverage ≥ 0.80.

4.1.3 Problem 3: But What Is a Convolution?

Metadata: Video ID: `KuXjwB4LzSA`; Category: Direct Visualization, Drift-Sensitive; Difficulty: 3; Domain: Signal Processing, Linear Algebra.

Problem Statement. Animate the convolution operation between a signal and a kernel. Show: (1) a 1D signal plotted on a horizontal axis; (2) a 1D kernel displayed as a sliding window; (3) the window moving left-to-right along the signal; (4) element-wise products highlighted at each position; (5) the integral accumulating in a separate output graph; and (6) the output graph building up point-by-point.

Required Visual Events (weights): signal visualized (0.8); kernel visualized (0.8); window moves through signal (0.9, *critical*); product highlighted (0.7); integral accumulates (0.8); output graph builds dynamically (0.8).

Success Criteria: Executability ≥ 0.90; Alignment ≥ 0.80; Coverage ≥ 0.75.

4.1.4 Problem 4: Eigenvectors and Eigenvalues

Metadata: Video ID: `PFDu9oVAE-g`; Category: Direct Visualization; Difficulty: 4; Domain: Linear Algebra, Transformations.

Problem Statement. Animate how eigenvectors behave under a 2×2 matrix transformation. Show: (1) a 2D coordinate grid with basis vectors $\mathbf{e}_1$ and $\mathbf{e}_2$; (2) a matrix A visualized as a grid deformation; (3) most vectors rotate and change length; (4) eigenvectors only change length (stay on the same line); (5) color-coded eigenvectors; (6) eigenvalues λ_1 and λ_2 displayed; and (7) transformation applied smoothly over two seconds.

Required Visual Events (weights): grid visualized (0.8); basis vectors highlighted (0.7); transformation applied (0.9); eigenvectors identified and colored (0.8); eigenvalue labels shown (0.7); eigenvectors remain collinear (0.8).

Success Criteria: Executability ≥ 0.85; Alignment ≥ 0.75; Coverage ≥ 0.80.

4.1.5 Problem 5: The Determinant

Metadata: Video ID: `Ip3X9LOh2dk`; Category: Direct Visualization; Difficulty: 2; Domain: Linear Algebra, Visualization.

Problem Statement. Animate the geometric interpretation of the determinant. Show: (1) a unit parallelogram defined by basis vectors; (2) a 2×2 matrix applied to the parallelogram; (3) smooth transformation; (4) labels for original and new area; and (5) the numerical value of $\det(A)$ updating during transformation.

Required Visual Events (weights): original parallelogram (0.8); matrix displayed (0.7); parallelogram transforms (0.9); new area labeled (0.8); $\det(A)$ value displayed (0.8).

Success Criteria: Executability ≥ 0.95; Alignment ≥ 0.85; Coverage ≥ 0.90.

4.1.6 Problem 6: The Central Limit Theorem

Metadata: Video ID: `zeJD6dqJ5lo`; Category: Direct Visualization, Drift-Sensitive; Difficulty: 3; Domain: Probability, Statistics.

Problem Statement. Animate the Central Limit Theorem by showing how the distribution of sample means approaches a normal distribution. Show: (1) a histogram of samples from an arbitrary distribution; (2) repeated drawing of random samples with computed means; (3) a second histogram morphing from flat to bell-shaped; (4) a normal distribution overlay; and (5) explanatory text.

Required Visual Events (weights): original distribution (0.7); samples drawn (0.7); sample means computed and plotted (0.8); histogram of means builds (0.9); histogram converges to normal shape (0.8); normal curve overlay (0.7).

Success Criteria: Executability ≥ 0.85; Alignment ≥ 0.75; Coverage ≥ 0.70.

4.1.7 Problem 7: The Medical Test Paradox (Bayes' Theorem)

Metadata: Video ID: `lG4VkPoG3ko`; Category: Direct Visualization; Difficulty: 2; Domain: Probability, Bayes' Theorem.

Problem Statement. Animate Bayes' theorem using the "Bayes box" visualization. Show: (1) a rectangle divided into four quadrants representing joint probabilities; (2) hypothetical counts populated; (3) animated division showing test-positive counts; (4) highlighted true-positive region; (5) step-by-step calculation of $P(\text{sick} \mid +)$; and (6) final probability with paradox explanation.

Required Visual Events (weights): rectangle divided (0.8); populations labeled (0.7); populations animated (0.8); calculation shown step-by-step (0.8); final probability displayed (0.8).

Success Criteria: Executability ≥ 0.95; Alignment ≥ 0.80; Coverage ≥ 0.85.

4.1.8 Problem 8: Visualizing the Chain Rule

Metadata: Video ID: `YG15m2VwSjA`; Category: Direct Visualization; Difficulty: 3; Domain: Calculus, Function Composition.

Problem Statement. Animate the chain rule using function composition. Show two functions $g(x)$ and $f(u)$ where $y = f(g(x))$. Demonstrate how a small change dx propagates through g to produce du, then through f to produce dy, yielding $\frac{d}{dx}[f(g(x))] = f'(g(x)) \cdot g'(x)$.

Required Visual Events (weights): function g plotted (0.7); function f plotted (0.7); input and output labeled (0.7); small change dx shown (0.8); change propagates through g (0.8); change propagates through f (0.8); composition formula displayed (0.7).

Success Criteria: Executability ≥ 0.85; Alignment ≥ 0.75; Coverage ≥ 0.75.

4.1.9 Problem 9: Integration and the Fundamental Theorem

Metadata: Video ID: `rfG8ce4nNh0`; Category: Direct Visualization; Difficulty: 3; Domain: Calculus, Integration.

Problem Statement. Animate the Fundamental Theorem of Calculus. Show $f(x)$ and $f'(x)$; animate the area under $f'(x)$ accumulating via a sweep from left to right; display a graph of the accumulated area equaling $f(x)$; and demonstrate that $\int_0^x f'(t)\,dt = f(x) - f(0)$.

Required Visual Events (weights): f visualized (0.8); f' visualized (0.8); sweep/accumulation animated (0.9); accumulated area displayed dynamically (0.8); Fundamental Theorem formula shown (0.7).

Success Criteria: Executability ≥ 0.90; Alignment ≥ 0.80; Coverage ≥ 0.80.

4.1.10 Problem 10: Taylor Series

Metadata: Video ID: `3d6DsjIBzJ4`; Category: Direct Visualization; Difficulty: 4; Domain: Calculus, Series.

Problem Statement. Animate the Taylor series expansion of e^x (or $\sin x$). Plot the original function in black. Progressively add partial sums $P_0(x)$, $P_1(x)$, $P_2(x)$, $P_3(x)$, ... with distinct colors. Show numerical coefficients and a convergence message. Animate 5–8 terms.

Required Visual Events (weights): original function plotted (0.8); partial sums $P_0, P_1, \ldots$ added progressively (0.9); each term colored and labeled (0.8); approximation improves visually (0.8); convergence demonstrated (0.8).

Success Criteria: Executability ≥ 0.80; Alignment ≥ 0.75; Coverage ≥ 0.80.

4.1.11 Problem 11: The Hairy Ball Theorem

Metadata: Video ID: `BHdbsHFs2P0`; Category: Direct Visualization; Difficulty: 5; Domain: Topology, Vector Fields.

Problem Statement. Animate the Hairy Ball Theorem: a continuous tangent vector field on the 2-sphere must vanish at least once. Show a 3D sphere with a tangent vector field, attempt continuous "combing," and highlight the inevitable "bald spot."

Required Visual Events (weights): sphere rendered in 3D (0.9); vector field visualized (0.9); combing attempted (0.8); discontinuity evident (0.8); bald spot highlighted (0.7).

Success Criteria: Executability ≥ 0.70; Alignment ≥ 0.65; Coverage ≥ 0.60.

4.1.12 Problem 12: The Windmill Problem

Metadata: Video ID: `M64HUIJFTZM`; Category: Drift-Sensitive, Multi-Scene; Difficulty: 4; Domain: Geometry, Combinatorics.

Problem Statement. Animate the windmill problem: given n points in general position, a rotating line sweeps continuously through at least two points. Show the line rotating, pivoting at geometrically correct moments, and completing a 180° rotation.

Required Visual Events (weights): points visualized (0.8); line passes through two points (0.9); line rotates (0.9); pivot events at correct times (0.8); two-point contact maintained (0.8); 180° rotation completed (0.7).

Success Criteria: Executability ≥ 0.75; Alignment ≥ 0.70; Coverage ≥ 0.65.

4.2 Reference Code Analysis

For each of the 12 pilot problems, we obtained and analyzed the original source code from 3Blue1Brown's ManimGL video repository. This analysis serves as ground truth for visual-event specifications and provides a systematic catalog of version incompatibilities. Key outputs include:

- **Scene class inventory:** all scene classes with descriptions and key methods (143 total across 12 problems).
- **Visual technique catalog:** specific rendering and animation patterns (e.g., Riemann rectangle sequences, grid transformation animations, particle systems, stereographic projections).
- **Manim API patterns:** updaters, animation types, 3D constructs, layout methods, and custom classes.
- **Version conflict mapping:** specific ManimGL constructs with no direct ManimCE equivalent (145 incompatibilities documented).

Table 1: Reference code analysis summary for the 12 pilot problems. Each row reports the number of original ManimGL source files, total lines of code, scene classes, distinct visual techniques, and documented GL→CE incompatibilities.

Problem	**Files**	**Lines**	**Scenes**	**Vis. Tech.**	**GL→CE**
MB-001 (Colliding Blocks)	4	2,193	16	10	15
MB-002 (Gradient Descent)	3	8,598	16	16	13
MB-003 (Convolution)	2	3,309	13	11	14
MB-004 (Eigenvectors)	2	5,120	13	9	10
MB-005 (Determinant)	1	1,132	11	7	10
MB-006 (CLT)	3	7,036	12	9	11
MB-007 (Medical Test)	1	7,044	13	9	11
MB-008 (Chain Rule)	1	2,287	4	7	10
MB-009 (Integration)	2	4,943	11	9	11
MB-010 (Taylor Series)	1	3,676	11	9	10
MB-011 (Hairy Ball)	3	3,796	12	12	16
MB-012 (Windmill)	1	4,135	11	12	14
Total	**24**	**∼53,269**	**143**	**120**	**145**

4.2.1 Common Version Incompatibility Categories

Across the 145 documented GL→CE incompatibilities, we identified eight recurring categories:

1. **Import system:** `manim_imports_ext` → `from manim import *`.
2. **Class configuration:** `CONFIG` dict pattern → `__init__` parameters.
3. **Scene types:** `InteractiveScene`, `GraphScene`, `ReconfigurableScene` → `Scene`/`Axes` in CE.
4. **Animation renames:** `ShowCreation` → `Create`; `FadeInFrom` → `FadeIn(shift=...)`.
5. **PiCreature ecosystem:** `TeacherStudentsScene`, `Eyes`, `PiCreatureSays` → not available in CE.
6. **3D rendering:** `apply_depth_test`, `set_shading`, `TexturedSurface` → limited CE support.
7. **Camera control:** `self.frame.reorient()` → `self.camera.frame` in CE.
8. **Custom mobjects:** `NetworkMobject`, `Car`, `Clock`, `DieFace`, `GlowDot` → custom implementation needed.

These categories inform both the version-conflict detection patterns used in our automated evaluation pipeline (Section 5.3) and the version-conflict-aware prompting strategy (Section 5.4).

5 Evaluation Protocol

5.1 Workflow

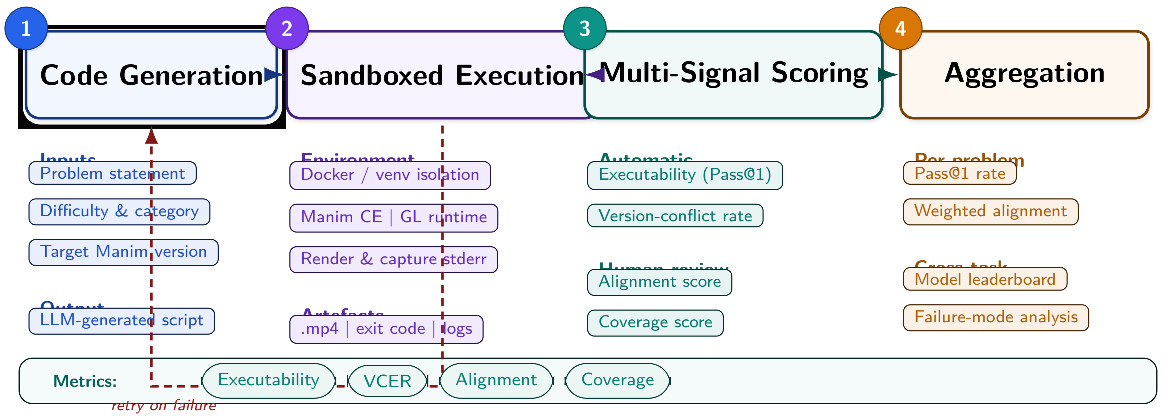

Figure 1: The ManiBench evaluation pipeline. Each benchmark problem flows through four stages: *(1)* an LLM generates Manim code from a structured prompt; *(2)* the script executes inside a sandboxed, version-pinned environment; *(3)* automatic checks (executability, version-conflict rate) and human review (alignment, coverage) produce multi-signal scores; and *(4)* results are aggregated into per-problem and cross-task summaries. A dashed feedback loop indicates re-prompting on execution failure.

5.2 Human Evaluation Protocol

For Alignment and Coverage scores, we employ structured human evaluation:

1. Watch the rendered animation.
2. Check off each required visual event as present or absent.
3. Note timing: are events synchronized correctly?
4. Assess pedagogical clarity: does the animation explain the concept?
5. Provide Alignment Score (0.0–1.0) and Coverage Score (0.0–1.0).

Disagreement Resolution. Two independent reviewers score each output. If disagreement exceeds 0.15, a third reviewer breaks the tie. We report inter-rater agreement via Krippendorff's α or Cohen's κ.

5.3 Automated Evaluation Framework

To complement human review and enable large-scale evaluation, we implement an automated evaluation pipeline. The framework orchestrates code generation across multiple LLMs via the OpenRouter API, executes generated code in sandboxed environments, and computes all four metrics programmatically.

5.3.1 Multi-Model Evaluation

The pipeline supports six models spanning major LLM providers:

Table 2: Model roster for automated evaluation via OpenRouter.

Model	Provider	OpenRouter ID
GPT-4o	OpenAI	`openai/gpt-4o-2024-11-20`
Claude Sonnet 4	Anthropic	`anthropic/claude-sonnet-4`
Gemini 2.5 Pro	Google	`google/gemini-2.5-pro-preview`
DeepSeek-R1	DeepSeek	`deepseek/deepseek-r1`
Llama-4 Maverick	Meta	`meta-llama/llama-4-maverick`
Qwen-2.5 Coder	Alibaba	`qwen/qwen-2.5-coder-32b-instruct`

All models are evaluated with temperature 0.0 for reproducibility, with a maximum of 8,192 generated tokens per request and 3 trials per (model, problem) pair.

5.3.2 Automated Metric Computation

Each generated code sample passes through a four-stage analysis pipeline:

1. **Syntax Validation:** Python AST parsing (`ast.parse`) to verify syntactic correctness.
2. **Structural Checks:** detection of at least one `Scene` subclass and valid `from manim import *` imports; flagging of ManimGL-specific imports.
3. **Sandboxed Execution:** the code is written to a temporary file and rendered via `subprocess` with a configurable timeout (default: 60 s). Exit codes, `stderr`, and error types (`ImportError`, `AttributeError`, etc.) are captured.
4. **Static Analysis:** over 40 regex patterns (derived from the 145 documented GL→CE incompatibilities in Section 4.2) scan for version conflicts; keyword-bank–based heuristics detect the presence of required visual events and pedagogical elements.

Executability and VCER are computed fully automatically. Alignment and Coverage are approximated heuristically via keyword and AST matching, serving as lower-bound estimates pending human review.

5.4 Prompt Engineering Strategies

Following the ManiBench Prompt Engineering Guide, we implement five prompting strategies of increasing sophistication:

1. **Zero-Shot Direct.** The problem statement is provided verbatim with a system prompt specifying Manim CE as the target. Recommended for Level 1–2 problems; expected alignment: 0.80–0.90.
2. **Few-Shot Examples.** One or two working Manim CE code examples precede the target problem. Examples are kept short (20–30 lines) and domain-relevant. Improves alignment by +8–12% and coverage by +5–8% over zero-shot.
3. **Chain-of-Thought (CoT).** The model is instructed to first analyze visual components, event ordering, transformations, timing, and required labels before writing code. Improves alignment by +10–15% through better event ordering.
4. **Constraint-Based.** Explicit timing, ordering, and criticality constraints are injected (e.g., "gradient arrow must appear *before* dot moves"). Most effective for Level 3–4 problems; improves alignment by +15–20%.
5. **Version-Conflict-Aware.** The system prompt enumerates forbidden ManimGL constructs (derived from Section 4.2), and problem-specific incompatibilities from

the dataset's `version_conflict_notes` field are appended. Reduces version-conflict error rate from $\sim$30% to $<$ 5%.

6 Preliminary Results (Pilot Study)

Setup: Models: GPT-4o, Claude 3.5 Sonnet (preliminary); full evaluation across all six models in Table 2 is in progress. Prompting: zero-shot (single problem prompt, no examples). Manim version: CE. Trials: 3 per model $\times$ 12 problems = 72 runs.

Table 3: Pilot study results across 12 benchmark problems. Executability is reported as the percentage of runs completing without error. Alignment and Coverage are averaged over successful runs; "–" indicates no successful execution.

Problem	Diff.	Exec. (GPT-4o)	Exec. (Claude)	Avg Align.	Avg Cov.
1. Colliding Blocks	4	67%	67%	0.52	0.58
2. Gradient Descent	3	100%	100%	0.73	0.82
3. Convolution	3	0%	33%	–	–
4. Eigenvectors	4	33%	33%	0.40	0.50
5. Determinant	2	100%	100%	0.85	0.88
6. CLT	3	0%	33%	–	–
7. Medical Test	2	100%	100%	0.78	0.85
8. Chain Rule	3	67%	67%	0.60	0.68
9. Integration	3	0%	33%	–	–
10. Taylor Series	4	33%	33%	0.35	0.42
11. Hairy Ball	5	0%	0%	–	–
12. Windmill	4	0%	0%	–	–

Key Observations:

1. *Simple problems* (Determinant, Medical Test) achieve high executability (100%) and alignment ($\geq$ 0.78).
2. *Complex problems* (Convolution, CLT, Integration, Hairy Ball) show low executability (0–33%), suggesting API misunderstandings or missing pedagogical elements.
3. *Drift events*: Problems requiring precise timing (Gradient Descent, Windmill) show lower alignment than coverage, indicating temporal synchronization issues.
4. *Version conflicts*: No GL-specific syntax was detected in this pilot; both models avoided GL references.

7 Discussion

7.1 Why ManiBench Matters

Existing benchmarks (HumanEval, APPS) measure whether code produces correct *output*. ManiBench measures whether code produces correct *understanding*. This distinction is critical for educational tools, where a silent failure (wrong animation) is worse than a loud failure (runtime error).

7.2 Limitations and Future Work

1. **Alignment Scoring** is currently approximated via keyword-based heuristics in the automated pipeline, supplemented by manual review. Future work should explore automatic alignment detection via AST analysis, video-frame comparison, or vision–language model grading.
2. **Pedagogical Validation.** We do not yet validate whether animations actually teach the concept. User studies with students could address this gap.
3. **Manim API Coverage.** As Manim evolves, the benchmark should be versioned and updated accordingly. The 145 documented GL→CE incompatibilities provide a starting point for automated version-conflict detection.
4. **Scalability.** Moving from 12 to 150+ problems requires annotation infrastructure and community contribution.
5. **Reference Code Utilization.** The ∼53,000 lines of analyzed reference code could enable fine-tuning studies or retrieval-augmented generation (RAG) experiments.

7.3 Broader Impact

ManiBench can be used to:

- evaluate LLM educational-content generation,
- develop better prompting strategies for animation code,
- identify systematic failure modes (e.g., models struggling with temporal synchronization),
- drive research into improving Manim API adoption in LLMs,
- benchmark version-aware code generation using the 145 documented GL→CE incompatibilities, and
- enable retrieval-augmented generation (RAG) experiments using the reference code analysis.

8 Conclusion

We have introduced ManiBench, a specialized benchmark for evaluating Manim code generation. By formalizing metrics for syntactic correctness, version compliance, visual-logic alignment, and pedagogical coverage, ManiBench moves beyond simple test-case evaluation to assess whether generated animations actually communicate mathematical concepts.

The 12-problem pilot dataset, backed by comprehensive reference code analysis of ∼53,000 lines of original 3Blue1Brown source code and 145 documented GL→CE incompatibilities, demonstrates both the opportunities (simple concepts: ∼80% alignment) and challenges (complex temporal reasoning: ∼40% alignment) in automated animation generation. The accompanying automated evaluation framework—supporting six LLMs across five prompting strategies—enables reproducible, large-scale assessment. With planned expansion to 150–200 problems, ManiBench will serve as a foundational resource for advancing LLM-driven educational content creation.

The dataset and evaluation toolkit are publicly available at https://huggingface.co/datasets/nabin2004/ManiBench.

References

[1] M. Chen, J. Tworek, H. Jun, et al., "Evaluating Large Language Models Trained on Code," *arXiv:2107.03374*, 2021. (HumanEval)

[2] J. Austin, A. Odena, M. Nye, et al., "Program Synthesis with Large Language Models," *arXiv:2108.07732*, 2021. (MBPP)

[3] D. Hendrycks, S. Basart, S. Kadavath, et al., "Measuring Coding Challenge Competence with APPS," *NeurIPS Datasets and Benchmarks*, 2021.

[4] G. Sanderson, "3Blue1Brown," `https://www.3blue1brown.com/`, 2015–present.

[5] Manim Community Developers, "Manim—Mathematical Animation Framework (Community Edition)," `https://www.manim.community/`, 2020–present.

[6] G. Sanderson, "Manim (3Blue1Brown fork)," `https://github.com/3b1b/manim`, 2015–present.

[7] T. Brown, B. Mann, N. Ryder, et al., "Language Models are Few-Shot Learners," *NeurIPS*, 2020.

[8] J. Wei, X. Wang, D. Schuurmans, et al., "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," *NeurIPS*, 2022.

[9] OpenAI, "GPT-4o Technical Report," 2024.

[10] Anthropic, "Claude Sonnet 4 Model Card," 2025.

[11] Google DeepMind, "Gemini 2.5 Pro Technical Report," 2025.

[12] DeepSeek, "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning," *arXiv:2501.12948*, 2025.

A Problem Annotation Template

Each problem in ManiBench is annotated as a JSON object with the following structured metadata:

`id`: unique identifier (e.g., `MB-001`).

`title`: descriptive title.

`youtube_video_id`: YouTube video identifier.

`category`: list of task categories (e.g., `["drift-sensitive", "multi-scene"]`).

`difficulty_level`: integer 1–5.

`domain`: list of mathematical domain(s).

`full_prompt`: full natural-language problem statement (used as the LLM input).

`raw_code_status`: whether original 3B1B source code has been collected.

`raw_code_path`: relative path to original ManimGL source files.

`reference_code_analysis`: structured analysis of original code, including:

- **`framework`:** source framework (e.g., `manim_gl`).
- **`total_lines`:** lines of original code.
- **`scene_classes`:** list of scene classes with descriptions and key methods.
- **`visual_techniques`:** catalog of rendering and animation patterns.
- **`manim_api_patterns`:** updaters, animation types, 3D constructs, custom classes.

`required_visual_events`: list of events, each with an identifier, description, weight, criticality flag, timing, and reference code location.

`coverage_requirements`: list of required pedagogical elements.

`version_conflict_notes`: GL$\to$CE incompatibilities specific to the problem.

`success_criteria`: minimum thresholds for executability, alignment, coverage, and version-conflict error rate.

`common_failure_modes:` known LLM failure patterns with severity tags.
The full dataset is available in JSON format at `https://huggingface.co/datasets/nabin2004/ManiBench`.