Do Language Models Follow Their Own Plans? Plan-to-Artifact Faithfulness in Music and Code Generation

Anonymous ACL submission

Abstract

Large language models are increasingly prompted to produce an explicit plan before generating a complex artifact such as a program or symbolic score. Yet the final artifact may silently omit, contradict, or drift from the plan's commitments. We study this failure mode as *plan-to-artifact unfaithfulness*, treating plans as auditable specifications whose commitments the final artifact should implement. We introduce PACT (**P**lan–**A**rtifact **C**onsistency **T**esting), a framework that elicits structured plans, generates artifacts, verifies plan–artifact consistency via cross-model monitors, and uses detected mismatches to guide revision. We instantiate PACT in code generation (LiveCodeBench, BigCodeBench-Hard) and symbolic music generation, evaluating with three frontier models. Cross-model monitors estimate that 32–79% of code artifacts and 93–100% of music artifacts contain at least one plan violation under our protocol, with independent rule-based checks corroborating these trends. Notably, 31.6% of code artifacts pass all held-out tests while violating their stated plans; standard test-based evaluation does not capture this dimension of quality. Plan-aware revision outperforms generic iterative feedback, reducing severity-weighted mismatches by 15–59%.

1 Introduction

Explicit planning has become a common strategy for eliciting stronger performance from large language models (LLMs). A model may first outline an algorithm before writing code, decompose a reasoning problem into subtasks, or sketch the key, structure, and chord progression before composing a piece of music. Such plans are useful: prior work shows that planning can improve reasoning and code generation (Wang et al., 2023; Jiang et al., 2024; Wang et al., 2025a; Lei et al., 2025). Yet a visible plan also creates an expectation: the final artifact should implement the commitments that the plan makes.

In practice, this expectation often fails. A code-generation model may state that it will use breadth-first search with a visited set, but generate recursive depth-first search and omit the unreachable-target case. A music-generation model may state that it will write a balanced 4+4 antecedent–consequent period ending in a half cadence, but produce a 5+3 phrase that cadences on tonic. These outputs may compile and look reasonable, but they are unfaithful to the model's own specification.

We call this phenomenon **plan-to-artifact unfaithfulness**: the final generated artifact omits, contradicts, or changes commitments made in an explicit model-generated plan. This differs from chain-of-thought (CoT) faithfulness, which asks whether a reasoning trace reflects the model's true causal process (Turpin et al., 2023; Lanham et al., 2023). We instead treat plans operationally, as user-visible commitments. Regardless of whether the model actually relies on the plan during generation, an artifact that violates it can mislead users and reduce trust in plan-guided systems.

To study this problem, we introduce PACT (**P**lan–**A**rtifact **C**onsistency **T**esting), a framework that elicits structured plans, generates artifacts, verifies plan–artifact consistency via cross-model monitors, and uses detected mismatches to guide revision. We instantiate PACT in two structurally different domains (code generation, where plans specify algorithms, data structures, and edge cases, and symbolic music generation, where plans specify harmony, phrase structure, and cadences) to separate the general phenomenon from domain-specific quality judgments.

Our investigation is organized around four questions. How often do artifacts violate their plans (**RQ1**)? Can cross-model monitors detect these mismatches (**RQ2**)? Does plan-specific feedback reduce them more than generic iteration (**RQ3**)?

And do the patterns generalize across domains (**RQ4**)? We evaluate PACT across three model families on two code benchmarks and 30 music tasks.

2 Related Work

Planning and CoT faithfulness. Explicit planning has become a standard ingredient in LLM pipelines. Plan-and-Solve (Wang et al., 2023), self-planning (Jiang et al., 2024), PlanSearch (Wang et al., 2025a), and LPW (Lei et al., 2025) all show that generating a plan before code improves downstream correctness, while Think Anywhere (Jiang et al., 2026) interleaves reasoning within code itself. These methods evaluate plans by whether the final output is correct, but correctness does not guarantee that the artifact actually follows the plan. A parallel line of work raises similar concerns for chain-of-thought reasoning: CoT prompting improves accuracy (Wei et al., 2022), yet the generated rationales may not reflect the model's true decision process (Turpin et al., 2023; Lanham et al., 2023), motivating methods that enforce causal consistency between reasoning steps and answers (Lyu et al., 2023; Swaroop et al., 2025). We ask a complementary question: even when the plan is visible and correct, does the artifact implement it?

Iterative refinement and cross-representation consistency. Iterative refinement (Self-Refine (Madaan et al., 2023), Reflexion (Shinn et al., 2023), ReAct (Yao et al., 2023), MAMM-Refine (Wan et al., 2025)) shows that LLM outputs improve with feedback, but the feedback is typically generic or task-level. PACT makes the target explicit: specific plan commitments. Several recent studies examine adjacent consistency and execution gaps. Sinha et al. (2025) show that LLMs fail at execution even when they can plan; Xue et al. (2025) identify claim–implementation mismatch in thinking code LLMs; Jin and Chen (2025) find that correction steps can induce overcorrection. IdentityChain (Min et al., 2024) and MPSC (Huang et al., 2024) verify that code, NL summaries, and test cases stay mutually consistent, while Han et al. (2022) use proof artifacts as co-training signals in theorem proving, sharing our intuition that intermediate representations should align with final outputs (and, coincidentally, the PACT acronym). Our framework targets a richer consistency relation: between structured plans specifying algorithms, invariants, and constraints, and the artifacts generated from them. For symbolic music, LLMs show increasing capability (Huang et al., 2019; Yuan et al., 2024; Wang et al., 2025b; Deng et al., 2024) but may encode music-theoretic concepts without reliably applying them (Wei et al., 2024; Zhou et al., 2024), making music a natural second domain for studying plan–artifact gaps.

3 Problem Definition

Let x denote a user task, p a model-generated explicit plan, and y the final generated artifact. A plan is a structured set of commitments:

$$p = \{c_1, c_2, \ldots, c_n\}, \tag{1}$$

where each commitment c_i specifies a single, independently falsifiable claim about the artifact. We extract commitments by decomposing each plan field into atomic claims: for example, "BFS with a visited set to avoid duplicates" yields two commitments (use BFS; maintain a visited set to avoid duplicates).

We define a realization function $R(y, c_i)$ that returns one of REALIZED, OMITTED, CONTRADICTED, or CHANGED, judging whether artifact y realizes commitment c_i. The artifact is **plan-faithful** if all commitments are realized:

$$\text{Faithful}(p, y) = \mathbb{I}\big[\forall c_i \in p,\ R(y, c_i) = \text{REALIZED}\big]. \tag{2}$$

A **plan–artifact mismatch** occurs when at least one commitment is omitted, contradicted, or changed. This definition separates plan quality (is p good for task x?), artifact correctness (does y satisfy x?), and plan-to-artifact faithfulness (does y implement p?). A poor plan can be faithfully implemented, and a correct artifact can be unfaithful to its stated plan. Our focus is the third relation. We categorize mismatches into seven types: omitted commitment, contradicted commitment, structural mismatch, constraint mismatch, validation mismatch, assumption drift, and overcorrection (Table 1). The first six apply to single-pass monitoring; overcorrection is detected across revision iterations. Each category is domain-agnostic; Table 1 illustrates each type with instances from music and code generation.

4 The PACT Framework

PACT contains four stages: planning, plan-conditioned generation, plan–artifact verification, and revision. The pipeline is domain-agnostic;

Mismatch type	Definition	Music example	Code example
Omitted commitment	Plan states a required element; artifact lacks it.	Plan calls for suspensions every 1–2 measures; score has none.	Plan says handle empty input; code indexes first element unguarded.
Contradicted commitment	Artifact implements the opposite or incompatible choice.	Plan says half cadence at bar 8; score resolves to tonic.	Plan says BFS; code uses recursive DFS.
Structural mismatch	High-level organization differs from stated structure.	Plan says 4+4 period; score has 5+3 grouping.	Plan says two-pass; code mixes passes.
Constraint mismatch	Artifact violates a stated constraint.	Plan says F minor; score modulates without preparation.	Plan claims $O(n)$; code uses nested scans.
Validation mismatch	Plan includes checks not implemented.	Plan says avoid parallel fifths; score has them.	Plan says validate bounds; code trusts indices.
Assumption drift	Artifact introduces unstated assumptions.	Plan assumes chorale texture; score becomes block chords.	Plan assumes arbitrary order; code assumes sorted.
Overcorrection†	Revision breaks one commitment while fixing another.	Style matching collapses into near-verbatim copying.	Patch hardcodes tests rather than preserving algorithm.

†Detected across iterations, not single-pass monitoring.

Table 1: A cross-domain taxonomy of plan–artifact mismatches.

what varies across instantiations is the plan schema, artifact format, and available correctness signals (Appendix A). Figure 1 illustrates the full pipeline.

4.1 Planning

Given task x, a planner LLM produces a structured plan p. We use domain-specific JSON schemas to make commitments explicit and annotatable. For code, the schema contains algorithm, data structures, invariants, edge cases, complexity, and input/output assumptions. For music, it contains key/mode, meter, phrase structure, harmonic plan, cadence locations, motivic material, development strategy, and style constraints.

Plan quality filter. A commitment is *specific* if it makes a falsifiable claim about the artifact. Before generation, a lightweight classifier counts specific commitments; if too few are found, the planner is re-prompted to be concrete (Appendix D). The structured JSON schemas already encourage specificity, but the filter catches cases where fields are filled with generic content.

4.2 Plan-Conditioned Generation

A generator LLM receives (x, p) and produces artifact y. The prompt instructs the generator to preserve the plan commitments and to avoid introducing unstated structural assumptions. We separate planning and generation so that plans remain user-visible and independently auditable.

4.3 Plan–Artifact Verification

Before monitoring, a commitment extractor decomposes the plan JSON into a numbered list of atomic, falsifiable commitments $\{c_1, \ldots, c_n\}$, following the segmentation rule in Section 3. This step is a separate LLM call, ensuring the monitor receives an explicit checklist. The extractor targets 8–20 commitments per plan, balancing coverage against monitor reliability; we cap the list if it exceeds the upper bound.

The monitor then receives the commitment list $\{c_1, \ldots, c_n\}$ and artifact y and returns a structured report:

$$M(p, y) = \{(c_i, r_i, e_i, s_i)\}_{i=1}^{n}, \qquad (3)$$

where r_i is the verdict (one of REALIZED, OMITTED, CONTRADICTED, or CHANGED), e_i is textual evidence from the artifact, and $s_i \in \{1, 2, 3\}$ indicates minor, major, or critical severity.

Cross-model monitoring. In our cross-evaluation design, the monitor always uses a different model family from the generator: every artifact is evaluated by two independent monitors. When both monitors agree on a mismatch, we treat the label as high-confidence; when they disagree, the disagreement surfaces potential false negatives and model-specific blind spots. This does not eliminate shared biases, but it raises the bar relative to single-model evaluation.

4.4 Revision

If any high-severity mismatch is found, a revision planner converts the monitor report into targeted instructions that identify the specific commitment violated and suggest a minimal fix. The generator then produces a revised artifact y', conditioned on the original task, the original plan, the previous artifact, and the mismatch report. This process

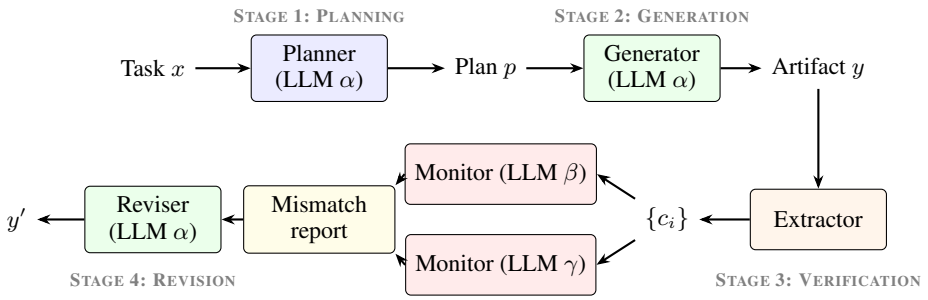

Figure 1: The PACT pipeline. A planner (LLM α) produces a structured plan p from which the same model generates artifact y. An extractor decomposes p into atomic commitments $\{c_i\}$, which two monitors from different model families (β, γ) independently check against y. Their consensus report drives targeted revision by the original generator.

can iterate for a fixed budget or until the monitor returns no high-severity mismatches.

5 Experimental Setup

5.1 Tasks

For **code**, we use 300 LiveCodeBench (Jain et al., 2025) problems (150 medium + 150 hard, stratified random, seed=42) and 148 BigCodeBench-Hard tasks (Zhuo et al., 2025); artifacts are Python functions evaluated against held-out tests. For **music**, we use 30 compositional prompts across 5 styles (Baroque chorale, Classical sonata, Romantic character piece, jazz lead sheet, and minimalist process piece); artifacts are ABC notation scores.

5.2 Models and Cross-Evaluation Design

We use three frontier models from different families: GPT-5.4 (OpenAI, 2026), Claude Sonnet 4.6 (Anthropic, 2026), and Gemini 3.5 Flash (Google DeepMind, 2026). Each model serves as both generator and monitor, but never for the same artifact: each artifact is monitored by the two models that did not generate it. For revision, the original generator revises its own artifact conditioned on the cross-model monitor reports. This design eliminates shared-backbone bias and provides a natural reliability proxy via inter-monitor agreement.

5.3 Baselines

To isolate what drives improvement, we compare PACT against systems that vary in whether they plan, what feedback they use, and whether they check faithfulness. **Direct** establishes a lower bound with no plan or revision. **Self-Refine** (Madaan et al., 2023) controls for additional compute: it receives the same number of revision turns as PACT, but with generic rather than plan-specific feedback. **LPW** (Lei et al., 2025) (code only) verifies that the *plan is correct* via visible tests but does not check whether the *artifact follows the plan*; the comparison is therefore limited to Pass@1, where PACT matches LPW. **PACT +T** adds test-execution feedback on top of PACT, jointly optimizing faithfulness and correctness.

5.4 Evaluation Protocol

Measuring plan–artifact consistency is harder than measuring task correctness: code test cases verify outputs but not whether the code follows its plan, and music lacks any compiler altogether. We combine LLM-based monitoring with multiple independent checks.

To assess **prevalence** (RQ1), we measure mismatches at two levels: how many individual commitments are violated (C-MR, commitment-level mismatch rate), and how many artifacts contain at least one violation (A-MR, artifact-level mismatch rate). Not all mismatches are equally serious, so we also report a severity-weighted score:

$$\text{SWM} = \frac{\sum_i s_i \cdot \mathbb{I}[R(y, c_i) \neq \text{REALIZED}]}{\sum_i s_i}, \quad (4)$$

where $s_i \in \{1, 2, 3\}$ is the monitor-assigned severity (minor, major, critical). Correct+Unfaithful (C+U) isolates the fraction of artifacts that pass all tests yet violate at least one commitment.

To assess **detection quality** (RQ2), we measure planted-mismatch recall on artifacts with known injected corruptions (Appendix E), noting that planted corruptions are likely more salient than natural ones. We complement this with deterministic, LLM-free checks on the roughly 20–30% of commitments that admit binary ground-truth answers: AST analysis for structural claims (e.g., queue instantiation when the plan specifies BFS),

Domain	Generator	N	C-MR	A-MR
Music	GPT-5.4	30	20.6%	93.3%
	Claude 4.6	30	59.1%	100.0%
	Gemini 3.5	30	51.2%	96.7%
LiveCode-Bench	GPT-5.4	300	2.9%	32.3%
	Claude 4.6	291	18.5%	79.0%
	Gemini 3.5	295	20.7%	72.2%
BigCode-Bench-H	GPT-5.4	148	1.5%	23.0%
	Claude 4.6	147	6.7%	53.1%
	Gemini 3.5	147	6.7%	57.8%

Table 2: Monitor-estimated mismatch prevalence across three benchmarks. C-MR: commitment-level mismatch rate. A-MR: artifact-level mismatch rate ($\geq$ 1 mismatch). The generator ranking GPT $<$ Claude $\approx$ Gemini is consistent across all three domains. N $<$ 300 for Claude/Gemini on LiveCodeBench reflects generation-time failures excluded before monitoring.

Evaluation	Category	Recall
Planted (code)	Contradicted	0.96
	Omitted	0.83
	Assumption drift	0.75
	Validation	0.67
	Constraint	0.53
	Structural	0.50
	Overall	0.74
Planted (music)	*Overall*	1.00
Rule-based (music)	GPT-5.4	67%
	Claude 4.6	77%
	Gemini 3.5	83%
Inter-monitor κ (natural)	Code	0.42–0.74
	Music	0.34–0.56

Table 3: Monitor reliability. Planted-mismatch recall on 60 code and 24 music corrupted artifacts (likely upper bounds); per-category breakdown for code shows contradicted commitments are easiest and structural mismatches hardest to detect. Rule-based checks on music independently confirm the generator ranking GPT $<$ Claude $<$ Gemini without LLM judgment (per-check breakdown in Table 15). Inter-monitor κ on unmodified model-generated outputs provides the primary reliability estimate.

complexity profiling, and edge-case execution for code; key signature, meter, measure count, and cadence verification via rule-based ABC parsing for music. This independent signal breaks the circularity of using LLM monitors both for prevalence (RQ1) and revision (RQ3). On unmodified model-generated artifacts, we report inter-monitor Cohen's κ between two independent model families; because consensus requires both monitors to flag a mismatch, our prevalence estimates are lower bounds.

To assess **repair and quality** (RQ3), we compare SWM before and after revision. Because faithfulness improvements are meaningful only if they do not degrade the artifact itself, we also track domain-specific quality metrics. For code, we report Pass@1 on the benchmarks' held-out test cases. For music, we report ABC-to-MIDI parse success rate as a measure of notation validity, and conduct a blind listening test with 20 raters who score each output on overall quality (does the piece sound coherent and well-constructed?) and task adherence (does it match what the prompt asked for?).

6 Results

6.1 RQ1: Plan–Artifact Mismatches Are Frequent

Table 2 reports monitor-estimated mismatch prevalence across all three benchmarks. LiveCodeBench tests algorithmic problem-solving; BigCodeBench-Hard tests software-engineering tasks involving library and API usage (pandas, sklearn, etc.); the music prompts span five compositional styles. Plan–artifact mismatches are frequent: even GPT-5.4, the generator with the lowest mismatch rate, produces code artifacts flagged with at least one mismatch 23–32% of the time depending on the benchmark. Claude Sonnet 4.6 and Gemini 3.5 Flash show substantially higher estimated rates (A-MR 53–79%). The generator ranking GPT $<$ Claude $\approx$ Gemini holds across both code benchmarks despite their different task types. Music artifacts are even less faithful (A-MR 93–100%), reflecting the difficulty of realizing structured harmonic and phrase-level commitments in ABC notation without compiler feedback.

6.2 RQ2: Monitor Reliability

Table 3 shows monitor reliability. On planted mismatches, monitors achieve 74% recall on code and 100% on music. Among code corruption categories, contradicted commitments are easiest to detect (96% recall), while structural mismatches and constraint violations are hardest (50–53%), suggesting monitors struggle most with holistic structural judgments. Rule-based checks on all 90 music artifacts independently flag 67% (GPT), 77% (Claude), and 83% (Gemini), producing the same generator ordering as the LLM monitors in Table 2. Inter-monitor agreement on the unmodified model-generated artifacts ranges from moderate (κ=0.42 for Claude–

System	GPT-5.4	Claude 4.6	Gemini 3.5	Avg.
Direct	69.0	18.3	38.0	41.8
Self-Refine	75.3	34.7	49.3	53.1
LPW	72.3	33.7	72.9	59.7
PACT	72.7	34.4	70.8	59.5
PACT +T	75.7	37.5	44.7	52.6

Table 4: Pass@1 (%) on LiveCodeBench. PACT matches LPW on correctness despite optimizing for plan faithfulness rather than test passage. Gemini's drop under PACT +T reflects overcorrection (see Section 7).

	SWM ↓		C+U (%)	
Generator	init.	rev.	PACT	+T
LiveCodeBench				
GPT-5.4	.037	.023	21.0	5.0
Claude 4.6	.195	.156	27.1	11.0
Gemini 3.5	.213	.177	46.8	10.5
Music				
GPT-5.4	.292	.121	–	–
Claude 4.6	.577	.471	–	–
Gemini 3.5	.570	.482	–	–

Table 5: Faithfulness metrics per generator. SWM: severity-weighted mismatch score before (init.) and after (rev.) revision. C+U: % of artifacts passing all tests but violating ≥ 1 plan commitment. Music has no test cases.

System	Generator	Parse	Rate
Direct	GPT-5.4	21/30	70.0%
	Claude 4.6	0/30	0.0%
	Gemini 3.5	21/28	75.0%
Self-Refine	GPT-5.4	19/30	63.3%
	Claude 4.6	0/30	0.0%
	Gemini 3.5	10/30	33.3%
PACT	GPT-5.4	26/30	86.7%
	Claude 4.6	16/30	53.3%
	Gemini 3.5	11/30	36.7%
By system			
	Direct	42/88	47.7%
	Self-Refine	29/90	32.2%
	PACT	53/90	**58.9%**

Table 6: ABC-to-MIDI parse success rate, an LLM-independent measure of notation validity. PACT produces the most parseable scores. Claude goes from 0% (Direct) to 53.3% (PACT).

System	Quality	Task Adh.	Plan F.
Direct	2.95 ± 1.05	2.85 ± 1.08	–
Self-Refine	2.62 ± 1.10	2.70 ± 1.05	–
PACT	$\mathbf{3.18} \pm 0.98$	$\mathbf{3.25} \pm 1.00$	2.95 ± 1.12

Table 7: Human listening evaluation (20 raters, ∼600 ratings, 1–5 Likert). Quality: does it sound coherent and well-constructed? Task Adh.: does it match the prompt? Plan F.: does it follow the plan (PACT only).

Gemini) to substantial (κ=0.74 for Claude–GPT), so the prevalence estimates in Table 2 should be interpreted as consensus-based lower bounds.

6.3 RQ3: Monitor-Guided Revision

Faithfulness improves. Plan-aware revision reduces SWM by 15–59% across all generators and both domains (Table 5). The reductions are largest for GPT-5.4 on music (58.6%) and smallest for Gemini on music (15.4%). Since revision is driven by monitor feedback and evaluated by the same monitors, these gains could partly reflect artifacts fitting monitor expectations rather than genuine improvement. Two independent signals argue against this: Pass@1 on code (Table 4), measured by ground-truth test cases, is comparable or improved under PACT; and ABC-to-MIDI parse rates (Table 6), measured by a deterministic parser, improve from 47.7% (Direct) to 58.9% (PACT).

Quality does not degrade. PACT matches LPW on Pass@1 (Table 4) despite never seeing test cases during revision. For music, a blind listening test with 20 raters (Table 7) confirms PACT outperforms both baselines on quality (3.18 ± 0.98 vs. 2.95 Direct, 2.62 Self-Refine) and task adherence (3.25 ± 1.00 vs. 2.85, 2.70). Self-Refine degrades both metrics relative to Direct, suggesting that generic iteration without plan structure can harm output quality. Claude goes from 0% parseable under Direct to 53.3% under PACT (Table 6).

Correct but unfaithful. The C+U column in Table 5 reveals that on average 31.6% of code artifacts pass every held-out test yet violate at least one plan commitment (simple average across generators: 21.0%, 27.1%, 46.8%). Standard test-based evaluation does not capture this: a model can achieve high Pass@1 while diverging from its own design commitments. PACT +T reduces C+U to 5–11%, but at a cost for some generators; we examine this tension in Section 7.

6.4 RQ4: Cross-Domain Mismatch Patterns

Table 8 shows the category distribution across domains. Both domains share the same top-three mismatch categories (structural mismatch, omitted commitment, and constraint mismatch) but in different proportions: code generators most often deviate from the planned algorithm structure, while music generators most often violate stated key, me-

Category	Code	Music
Structural mismatch	804	238
Omitted commitment	713	214
Constraint mismatch	441	344
Contradicted commitment	205	103
Assumption drift	146	12
Validation mismatch	104	10

Table 8: Mismatch category counts across domains (single-pass monitoring; overcorrection is excluded as it is detected across revision iterations). Code and music share the same top-three categories but in different order: structural mismatches dominate in code, constraint mismatches in music.

	Code		Music	
Generator	N	Mean $\|c\|$	N	Mean $\|c\|$
GPT-5.4	469	19.0	30	19.9
Claude 4.6	460	12.9	31	18.2
Gemini 3.5	463	11.4	30	16.1

Table 9: Average commitments per plan by generator and domain. GPT-5.4 generates roughly 50% more commitments than Claude or Gemini in code; the gap narrows in music.

Difficulty	Generator	N	C-MR	A-MR
Medium	GPT-5.4	150	1.7%	25.3%
	Claude 4.6	148	10.4%	66.2%
	Gemini 3.5	150	14.6%	64.0%
Hard	GPT-5.4	150	4.0%	39.3%
	Claude 4.6	144	25.8%	91.7%
	Gemini 3.5	146	26.6%	80.1%

Table 10: LiveCodeBench A-MR and C-MR by difficulty. Hard problems show roughly 2–2.5× higher C-MR for every generator.

Generator	Monitor	Detect	κ
GPT-5.4	Claude 4.6	29.8%	0.42
	Gemini 3.5	21.5%	
Claude 4.6	GPT-5.4	77.2%	0.54
	Gemini 3.5	38.3%	
Gemini 3.5	GPT-5.4	68.9%	0.74
	Claude 4.6	41.7%	

Table 11: Detection rate and inter-monitor κ per generator. GPT-5.4 as monitor flags most aggressively; Gemini is most conservative.

ter, or harmonic constraints. Assumption drift is far more common in code (146 vs. 12), where generators silently add sorting or type-coercion assumptions not in the plan. Validation mismatch is nearly exclusive to code, as expected, since music plans rarely specify explicit validation checks.

7 Analysis

Plan specificity and faithfulness. GPT-5.4 generates the most commitments per plan (Table 9; 19.0 avg in code, 19.9 in music) yet achieves the lowest mismatch rate, while Claude (12.9, 18.2) and Gemini (11.4, 16.1) produce fewer commitments but higher mismatch rates. This does not necessarily imply that more commitments cause better faithfulness; models that plan more specifically may simply also execute more reliably.

Task difficulty and faithfulness. Difficulty strongly predicts unfaithfulness (Table 10): hard LiveCodeBench problems show 39–92% A-MR compared to 25–66% for medium, and C-MR is roughly 2–2.5× higher across all generators. Harder tasks produce plans with more interdependent commitments, increasing the chance that the generator drops or deviates from at least one.

Monitor asymmetry. Table 11 reveals systematic monitor asymmetry: GPT-5.4 flags mismatches most aggressively (68.9–77.2%), while Gemini is most conservative (21.5–38.3%). Two signals favor the sensitivity interpretation over miscalibration: rule-based checks independently confirm that GPT is the most faithful generator (Table 15), and inter-monitor κ is highest for Gemini's artifacts (0.74) and lowest for GPT's (0.42), consistent with GPT producing fewer, subtler mismatches that are harder for any monitor to detect.

The overcorrection problem. PACT +T reduces C+U dramatically (Table 5), but for Gemini the cost is severe: Pass@1 falls from 70.8% to 44.7% as the reviser breaks working code while attempting to satisfy plan commitments. GPT-5.4, by contrast, improves on both axes. The pattern is consistent: models with larger baseline faithfulness gaps must modify more code during joint revision, increasing the risk of introducing new bugs. This cautions against naively enforcing plan adherence for generators with weaker plan-execution capability, and suggests that adaptive revision, scaling feedback intensity to the generator's reliability, may be needed.

When deviation is not harmful. Not all plan violations degrade the artifact. A model may plan BFS but implement DFS, or promise $O(n)$ but use nested loops, yet the code passes all tests because neither algorithm choice nor complexity is directly tested. The issue is not deviation itself but *undis-*

Granularity	Generator	Avg. $\|c\|$	C-MR	A-MR
Coarse (3–7)	GPT-5.4	6.2	4.8%	29.6%
	Claude 4.6	6.3	14.7%	57.1%
	Gemini 3.5	6.6	13.4%	57.1%
Default (8–20)	GPT-5.4	16.7	3.1%	31.0%
	Claude 4.6	16.7	12.0%	75.0%
	Gemini 3.5	16.3	17.6%	87.0%
Fine (15–30)	GPT-5.4	26.5	4.0%	31.0%
	Claude 4.6	25.3	14.2%	72.4%
	Gemini 3.5	23.3	15.6%	89.3%

Table 12: Sensitivity of mismatch rates to commitment extraction granularity (30 tasks per generator). The ranking GPT $<$ Claude $\approx$ Gemini is stable; GPT's A-MR remains $\sim$31% regardless of extraction granularity.

closed deviation: if the model silently changes its approach, the user-visible plan becomes misleading. Systems should aim for disclosed faithfulness: following the plan, or transparently updating it when beneficial deviation occurs.

Robustness to extraction granularity. Since commitment extraction is LLM-based, mismatch rates could be an artifact of extraction granularity. Table 12 tests this by re-extracting commitments at three granularity targets (30 tasks per generator). The generator ranking GPT $<$ Claude $\approx$ Gemini is stable across all levels, and GPT's A-MR remains $\sim$31% whether 6 or 27 commitments are extracted. This suggests the prevalence estimates in Table 2 are robust to reasonable variation in how finely plans are decomposed.

8 Limitations

Our framework evaluates explicit structured plans only, not hidden CoT or unstructured reasoning. All code prevalence and repair claims rely on LLM monitor labels rather than human annotation; we mitigate this through cross-model consensus, planted-mismatch benchmarks, and rule-based checks (Sections 5.4–6.2), but these do not eliminate the possibility of shared blind spots across model families. Human annotation of a stratified code sample is a priority for future work. Planted corruptions may be more salient than natural deviations, overestimating monitor recall. The music listening test (30 tasks, 20 non-specialist raters) confirms that PACT outputs are preferred but remains small-scale. The commitment extraction step is LLM-based, though the sensitivity analysis in Section 7 shows stable trends across granularities. The Self-Refine comparison controls for additional compute, but a finer-grained ablation removing individual components (e.g., cross-model monitoring, the plan quality filter, or commitment extraction) would further isolate which parts of the pipeline drive the observed improvements. The monitor asymmetry reported in Section 7 (GPT-5.4 flags more aggressively than Gemini) warrants further investigation, including per-monitor precision estimates on natural outputs. Finally, our evaluation covers three model families; the extent to which these findings transfer to open-weight or smaller models remains to be tested.

9 Ethics Statement

If a system presents an explicit plan and the artifact silently diverges, users may over-rely on incorrect safety or validation claims. Conversely, rigid plan adherence may be harmful when deviation improves correctness. Systems should aim for *disclosed* faithfulness: following the plan or transparently updating it. Our music tasks use generic style prompts (e.g., "compose a Baroque chorale" or "write a jazz ballad") rather than specific copyrighted works. The listening evaluation was conducted with volunteer raters who provided informed consent; the study was reviewed by our institutional review board and determined exempt.

10 Conclusion

We introduced plan-to-artifact faithfulness as a distinct failure mode: LLM-generated artifacts can omit, contradict, or silently deviate from their own explicit plans. Our framework, PACT, detects these mismatches via cross-model monitors and uses them to guide revision. The central finding is that plan violations are common yet not captured by standard test-based evaluation: a substantial fraction of code artifacts pass all tests while violating their stated plans. Plan-aware revision reduces these violations significantly, outperforming generic iteration without degrading artifact quality. As plans increasingly serve as user-facing explanations in coding assistants and agentic systems, auditing whether artifacts follow them becomes increasingly relevant. Future work will pursue human validation of mismatch labels and systems that retroactively update plans when beneficial deviation occurs.

References

Anthropic. 2026. Introducing Claude Sonnet 4.6. https://www.anthropic.com/news/claude-sonnet-4-6. Accessed: May 2026.

Qixin Deng, Qikai Yang, Ruibin Yuan, Yipeng Huang, Yi Wang, Xubo Liu, Zeyue Tian, Jiahao Pan, Ge Zhang, Hanfeng Lin, and 1 others. 2024. ComposerX: Multi-agent symbolic music composition with LLMs. In *Proceedings of the 25th International Society for Music Information Retrieval Conference*, pages 669–679.

Google DeepMind. 2026. Gemini 3.5: Frontier intelligence with action. https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-3-5/. Accessed: May 2026.

Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W. Ayers, and Stanislas Polu. 2022. Proof artifact co-training for theorem proving with language models. In *Proceedings of the International Conference on Learning Representations*.

Baizhou Huang, Shuai Lu, Xiaojun Wan, and Nan Duan. 2024. Enhancing large language models in coding through multi-perspective self-consistency. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1429–1450.

Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Ian Simon, Curtis Hawthorne, Noam Shazeer, Andrew M Dai, Matthew D Hoffman, Monica Dinculescu, and Douglas Eck. 2019. Music transformer: Generating music with long-term structure. In *Proceedings of the International Conference on Learning Representations*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *Proceedings of the International Conference on Learning Representations*.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30.

Xue Jiang, Tianyu Zhang, Ge Li, Mengyang Liu, Taozhi Chen, Zhenhua Xu, Binhua Li, Wenpin Jiao, Zhi Jin, Yongbin Li, and Yihong Dong. 2026. Think anywhere in code generation. *arXiv preprint arXiv:2603.29957*.

Haolin Jin and Huaming Chen. 2025. Uncovering systematic failures of LLMs in verifying code against natural language specifications. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*, pages 3819–3823. NIER track.

Tamera Lanham, Anna Chen, Ansh Radhakrishnan, Benoit Steiner, Carson Denison, Danny Hernandez, Dustin Li, Esin Durmus, Evan Hubinger, Jackson Kernion, and 1 others. 2023. Measuring faithfulness in chain-of-thought reasoning. *arXiv preprint arXiv:2307.13702*.

Chao Lei, Yanchuan Chang, Nir Lipovetzky, and Krista A. Ehinger. 2025. Planning-driven programming: A large language model programming workflow. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12647–12684, Vienna, Austria. Association for Computational Linguistics.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. 2023. Faithful chain-of-thought reasoning. In *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 305–329, Nusa Dua, Bali. Association for Computational Linguistics.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36.

Marcus J. Min, Yangruibo Ding, Luca Buratti, Saurabh Pujar, Gail Kaiser, Suman Jana, and Baishakhi Ray. 2024. Beyond accuracy: Evaluating self-consistency of code large language models with IdentityChain. In *The Twelfth International Conference on Learning Representations*.

OpenAI. 2026. Introducing GPT-5.4. https://openai.com/index/introducing-gpt-5-4/. Accessed: May 2026.

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

Akshit Sinha, Arvindh Arun, Shashwat Goel, Steffen Staab, and Jonas Geiping. 2025. The illusion of diminishing returns: Measuring long horizon execution in LLMs. *arXiv preprint arXiv:2509.09677*.

Anand Swaroop, Akshat Nallani, Saksham Uboweja, Adiliia Uzdenova, Michael Nguyen, Kevin Zhu, Sunishchal Dev, Ashwinee Panda, Vasu Sharma, and Maheep Chaudhary. 2025. FRIT: Using causal importance to improve chain-of-thought faithfulness. *arXiv preprint arXiv:2509.13334*.

Miles Turpin, Julian Michael, Ethan Perez, and Samuel R Bowman. 2023. Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. In *Advances in Neural Information Processing Systems*, volume 36.

David Wan, Justin Chen, Elias Stengel-Eskin, and Mohit Bansal. 2025. MAMM-refine: A recipe for improving faithfulness in generation with multi-agent collaboration. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 9882–9901.

Evan Z. Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, William Song, Vaskar Nath, Ziwen Han, Sean M. Hendryx, Summer Yue, and Hugh Zhang. 2025a. Planning in natural language improves llm search for code generation. In *Proceedings of the International Conference on Learning Representations*. ICLR 2025 Spotlight.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, pages 2609–2634.

Yashan Wang, Shangda Wu, Jianhuai Hu, Xingjian Du, Yueqi Peng, Yongxin Huang, Shuai Fan, Xiaobing Li, Feng Yu, and Maosong Sun. 2025b. NotaGen: Advancing musicality in symbolic music generation with large language model training paradigms. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, pages 10207–10215.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837.

Megan Wei, Michael Freeman, Chris Donahue, and Chen Sun. 2024. Do music generation models encode music theory? In *Proceedings of the 25th International Society for Music Information Retrieval Conference*, pages 680–687.

Haoran Xue, Gias Uddin, and Song Wang. 2025. An empirical study of reasoning steps in thinking code LLMs. *arXiv preprint arXiv:2511.05874*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *Proceedings of the International Conference on Learning Representations*.

Ruibin Yuan, Hanfeng Lin, Yi Wang, Zeyue Tian, Shangda Wu, Tianhao Shen, Ge Zhang, Yuhang Wu, Cong Liu, Ziya Zhou, Liumeng Xue, Ziyang Ma, Qin Liu, Tianyu Zheng, Yizhi Li, Yinghao Ma, Yiming Liang, Xiaowei Chi, Ruibo Liu, and 13 others. 2024. ChatMusician: Understanding and generating music intrinsically with LLM. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 6252–6271, Bangkok, Thailand. Association for Computational Linguistics.

Ziya Zhou, Yuhang Wu, Zhiyue Wu, Xinyue Zhang, Ruibin Yuan, Yinghao Ma, Lu Wang, Emmanouil Benetos, Wei Xue, and Yike Guo. 2024. Can LLMs "reason" in music? an evaluation of LLMs' capability of music understanding and generation. In *Proceedings of the 25th International Society for Music Information Retrieval Conference*, pages 103–110.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2025. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *Proceedings of the International Conference on Learning Representations*. Code: https://github.com/bigcode-project/bigcodebench.

A Domain Instantiation Details

Table 13 summarizes how the PACT pipeline is instantiated in each domain. The key difference is the availability of external correctness signals: code has held-out test cases that measure task correctness (but not plan faithfulness), while music relies entirely on LLM-based monitoring supplemented by rule-based ABC-parsing checks.

B Plan Schemas

The planner is prompted to fill a domain-specific JSON schema. Each field corresponds to a class of commitments that the monitor will later verify. Fields are intentionally fine-grained to encourage falsifiable claims; the plan quality filter (Appendix D) catches cases where fields are filled with generic content.

B.1 Music Plan Schema

```
{ "key_mode": "...",
  "meter_tempo": "...",
  "instrumentation": "...",
  "phrase_structure": ["..."],
  "harmonic_plan": ["..."],
  "cadences": ["..."],
  "motivic_material": "...",
  "development_strategy": "...",
  "style_constraints": ["..."] }
```

B.2 Code Plan Schema

```
{ "algorithm": "...",
  "data_structures": ["..."],
  "invariants": ["..."],
  "edge_cases": ["..."],
  "complexity": "...",
  "input_output_assumptions": ["..."],
  "failure_cases": ["..."] }
```

Component	Symbolic music	Code
Task source	30 prompts, 5 styles $\times$ 6 each	300 LiveCodeBench (150 med + 150 hard)
Artifact format	ABC notation	Python function
Plan schema	Key/mode, meter, phrase structure, harmonic plan, cadences, motivic material, style constraints	Algorithm, data structures, invariants, edge cases, complexity, I/O assumptions
Monitor strategy	LLM-based + rule-based ABC parsing	LLM-based + deterministic checks
External correctness	Planted-mismatch P/R/F1; inter-monitor κ	Held-out test Pass@1

Table 13: Domain instantiations of the PACT framework.

C Monitor Prompt Template

The following template is used for all LLM-based monitoring calls. The monitor receives the extracted commitment list and the generated artifact, then returns a structured JSON report. The same template is used across both domains; domain-specific context (e.g., ABC notation conventions) is prepended as a system message.

```
You are a plan-artifact consistency monitor.
Given a structured PLAN and a generated
ARTIFACT, check each plan commitment.

For each commitment:
1. Quote the commitment.
2. Identify evidence in the artifact.
3. Assign one label: REALIZED, OMITTED,
   CONTRADICTED, CHANGED, or UNCLEAR.
4. If not REALIZED, assign a category:
   omitted commitment, contradicted
   commitment, structural mismatch,
   constraint mismatch, validation
   mismatch, assumption drift, or
   overcorrection.
5. Assign severity: minor/major/critical.
6. Suggest the smallest fix.

Return JSON only.
```

D Plan Quality Filter Details

The plan quality filter uses a lightweight LLM prompt to classify each field in the plan schema as *specific* or *vague*. A commitment is specific if it makes a falsifiable claim; examples of vague vs. specific:

- Vague: "use appropriate data structures" $\rightarrow$ Specific: "use a min-heap to maintain the k smallest elements"
- Vague: "write an interesting melody" $\rightarrow$ Specific: "use ascending stepwise motion in the antecedent, descending leap-step in the consequent"
- Vague: "handle edge cases" $\rightarrow$ Specific: "return -1 for empty input; clamp negative values to 0"

If a plan has fewer than k=5 specific commitments, the planner is re-prompted: "Your plan contains vague commitments that cannot be verified. For each field, provide a concrete, falsifiable claim about the artifact."

E Planted-Mismatch Corruption Operators

Table 14 lists the specific corruption operators used to construct the planted-mismatch benchmark. Each operator transforms a plan-faithful artifact into one with a known, labeled mismatch.

F Rule-Based Music Checks

Table 15 breaks down the rule-based checks by individual property. Generators fail in qualitatively different ways: GPT-5.4 and Claude always produce valid ABC notation but violate key constraints roughly 40% of the time, while Gemini produces invalid notation 43% of the time but achieves the highest key accuracy (90%) when the notation is parseable.

G Example Mismatches

Music. Plan: "Use a balanced 4+4 period; measure 4 should end with a half cadence and measure 8 with an authentic cadence." Artifact: the score contains five measures before the first cadence and resolves measure 4 to tonic. Label: structural mismatch and contradicted commitment.

Code. Plan: "Use BFS with a queue and mark nodes visited when enqueued to avoid duplicates. Return -1 if the target is unreachable." Artifact: the code uses recursive DFS, marks visited after recursion, and raises an exception for unreachable targets. Label: contradicted commitment (BFS$\rightarrow$DFS), validation mismatch (exception instead of sentinel return).

Category	Code corruption	Music corruption
Contradicted	Replace BFS with DFS (or vice versa); swap greedy for brute-force.	Change half cadence to authentic cadence (or vice versa).
Omitted	Delete edge-case guard clause.	Remove all suspension/passing-tone figures.
Structural	Merge two-pass algorithm into single confused pass.	Delete measure to break planned phrase symmetry.
Constraint	Insert nested loop to violate complexity claim.	Insert chromatic notes violating stated key constraint.
Validation	Remove bounds check or input validation.	Remove voice-leading constraints (introduce parallel fifths).
Assumption drift	Add sort at beginning, changing input assumptions.	Reduce chorale texture to block chords.

Table 14: Corruption operators for planted-mismatch evaluation.

Generator	Valid ABC	Key	Meter	Any
GPT-5.4	100%	63%	100%	67%
Claude 4.6	100%	60%	100%	77%
Gemini 3.5	57%	90%	57%	83%

Table 15: Rule-based checks on 30 music artifacts per generator (Direct condition). Valid ABC: notation parses successfully. Key/Meter: parsed notation matches the plan's stated key signature or time signature. Any: at least one rule-based check flags a mismatch.

Generator	SWM init.	SWM rev.
GPT-5.4	.020	.011
Claude 4.6	.090	.073
Gemini 3.5	.082	.063

Table 16: BigCodeBench-Hard SWM before and after plan-aware revision. Reductions mirror the LiveCodeBench pattern (Table 5).

H BigCodeBench-Hard Results

To verify that plan–artifact mismatches are not specific to algorithmic competitive programming, we evaluate PACT on all 148 tasks from BigCodeBench-Hard (Zhuo et al., 2025), a software-engineering benchmark requiring library and API usage (pandas, numpy, sklearn, etc.). Plans in this setting describe data processing pipelines, library calls, and I/O patterns rather than textbook algorithms. Prevalence results are included in Table 2.

Mismatch patterns on BigCodeBench-Hard differ qualitatively from LiveCodeBench. Overall mismatch rates are lower (A-MR 23–58% vs. 32–79%), likely because library-usage tasks have simpler control flow than algorithmic problems, producing plans with fewer interdependent commitments. Contradicted commitments more often involve library or API substitution (e.g., the plan specifies `pandas.groupby` but the code uses manual dictionary aggregation) rather than algorithm substitution. Assumption drift involves data format assumptions (column types, missing values) rather than input ordering. Structural mismatches arise when the plan describes a multi-stage pipeline but the code collapses stages. These differences confirm that plan–artifact unfaithfulness is not specific to algorithmic competitive programming.

After revision, SWM drops by 19–45% across generators (Table 16), mirroring the LiveCodeBench pattern.