SEMANTIC CONTRACTS FOR TRACEABLE NL-TO-CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models can generate code from natural-language problem statements, yet remain unreliable on constraint-intensive tasks. A failure may mean the model misread the task, or that it implemented a reasonable reading incorrectly, and the two call for different remedies. Requirement-alignment methods improve task understanding before generation; program-repair methods use post-generation feedback to fix implementation errors. The two are rarely connected by an explicit, stable semantic target, so repair has no fixed set of task obligations against which to compare successive programs.

We introduce a semantic-contract-based dual-loop framework that keeps the semantic target and the program as separate states, each updated by its own loop. The Semantic Alignment Loop (SAL) induces a structured specification and revises it field by field, admitting a revision only when a coverage–faithfulness–precision diagnostic and a Spec Alignment Score (SAS) show non-regression and grounded progress; the specification current at termination is frozen as a semantic contract. The Implementation Repair Loop (IRL) then selects an initial program under that contract and repairs it iteratively from public-test feedback and conservatively compiled property clauses. Because the contract is fixed before repair begins, semantic revisions and implementation repairs are recorded as separate state transitions.

On 1,055 LiveCodeBench release_v6 problems, Full Dual-Loop produces 401 complete-suite passes, a Final Pass Rate of 0.3801: 9.19 percentage points above Direct NL-to-Code and 3.98 points above the strongest shared-protocol external baseline. Final correctness improves while the evidence for task obligations and implementation transitions remains inspectable.

1 INTRODUCTION

Large language models have made natural-language-to-code generation a central problem in machine learning and software engineering. A natural-language task specifies input assumptions, output protocols, constraints, exceptional cases, and semantic conditions; a generated program is an executable implementation of those obligations. Existing systems improve reliability before generation through plans, pseudocode, retrieval, and requirement alignment (Jiang et al., 2024; Lei et al., 2025; Mishra et al., 2023; Chen et al., 2023; Li et al., 2024; Tian et al., 2025; Jia et al., 2025; Li et al., 2026), or after generation through self-feedback, execution, compiler signals, and tests (Madaan et al., 2023; Shinn et al., 2023; Zheng et al., 2024; Wang et al., 2022; Dou et al., 2024).

Task misunderstanding and implementation error nevertheless remain distinct failure sources. Empirical studies report semantic misinterpretations, omitted requirements, and missing boundary cases that cannot be reduced to local coding mistakes (Tambon et al., 2025; Wang et al., 2025; Jin & Chen, 2025). Requirement alignment and program repair usually operate as separate stages: the alignment result is not maintained as an explicit, stable target during repair, while repeatedly revising that target from execution failures makes task interpretation and implementation change together. In either case, repair lacks a consistent set of obligations against which successive programs can be compared.

We address this problem with a semantic-contract-based dual-loop framework that places an explicit state boundary between semantic alignment and implementation repair. SAL induces, diagnoses, and revises a structured specification. A candidate specification replaces the current state only after

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

passing an acceptance gate; when SAL terminates, the current specification is frozen as the semantic contract. IRL then changes only program state: initial candidate selection and every repair round use the same contract. This boundary makes semantic revisions and implementation transitions separately observable.

Concretely, SAL represents task obligations in six field groups and uses a coverage–faithfulness–precision profile together with SAS to localize revisions. Safety and improvement predicates determine whether a field-level patch replaces the current specification. IRL generates an initial candidate set from the problem and frozen contract, selects one program using public feedback, and repairs only that current program over subsequent rounds. Verifier feedback is complemented by executable property clauses conservatively compiled from the contract, while private tests remain held out until final evaluation.

The main contributions are:

- We formulate NL-to-code generation as two explicitly controlled state processes: SAL updates semantic state and freezes its accepted form before IRL updates implementation state.
- We develop a contract-conditioned mechanism that combines gated field-level revision, conservative property compilation, public-feedback-based initial selection, iterative repair, and an evidence-linked trace.
- We evaluate the framework on all 1,055 LiveCodeBench release_v6 problems, three backbones, and 162 compatible HumanEval+ tasks, observing a 3.98-point gain over the strongest shared-protocol external baseline on the full release.

2 Related Work

2.1 Intermediate representations

Planning and pseudocode methods separate solution organization from implementation (Jiang et al., 2024; Mishra et al., 2023). LPW generates a solution plan, verifies it on visible tests, and reuses the verified plan during code refinement (Lei et al., 2025). CodeT and AceCoder use generated tests or auxiliary artifacts to rank and refine candidates (Chen et al., 2023; Li et al., 2024), while NL in the Middle studies natural-language intermediates for code translation (Tai et al., 2025). These representations primarily describe how to solve or translate a task. Our specification instead records what every implementation must satisfy and remains available across candidate programs.

2.2 Requirement alignment and multi-stage generation

MuFix treats specification misunderstanding as a distinct pre-generation failure and repairs it using generated tests (Tian et al., 2025). SpecFix detects ambiguous problem descriptions from behavioral clusters of generated programs and repairs the description through contrastive specification inference (Jia et al., 2025). REA-Coder builds requirement-oriented questions and reference answers, then masks and recovers requirement fragments after generation; detected mismatches can return the system to requirement alignment (Li et al., 2026). AlphaCodium and MapCoder organize problem analysis, planning, implementation, and debugging into multi-stage or multi-agent workflows (Ridnik et al., 2024; Islam et al., 2024). Our distinction is the lifecycle of the accepted requirement representation: SAL finishes before IRL, and execution feedback thereafter changes programs without reopening the semantic contract.

2.3 Execution-guided repair and semantic failure analysis

Self-Refine and Reflexion use model feedback or verbalized experience to guide iterative revision (Madaan et al., 2023; Shinn et al., 2023); OpenCodeInterpreter, COMPCODER, and StepCoder incorporate execution, compiler, or unit-test signals (Zheng et al., 2024; Wang et al., 2022; Dou et al., 2024). Complementary analyses show that generated code frequently omits constraints, invents assumptions, or mishandles edge cases, and that models struggle to verify code against natural-language specifications (Tambon et al., 2025; Wang et al., 2025; Jin & Chen, 2025). These obser-

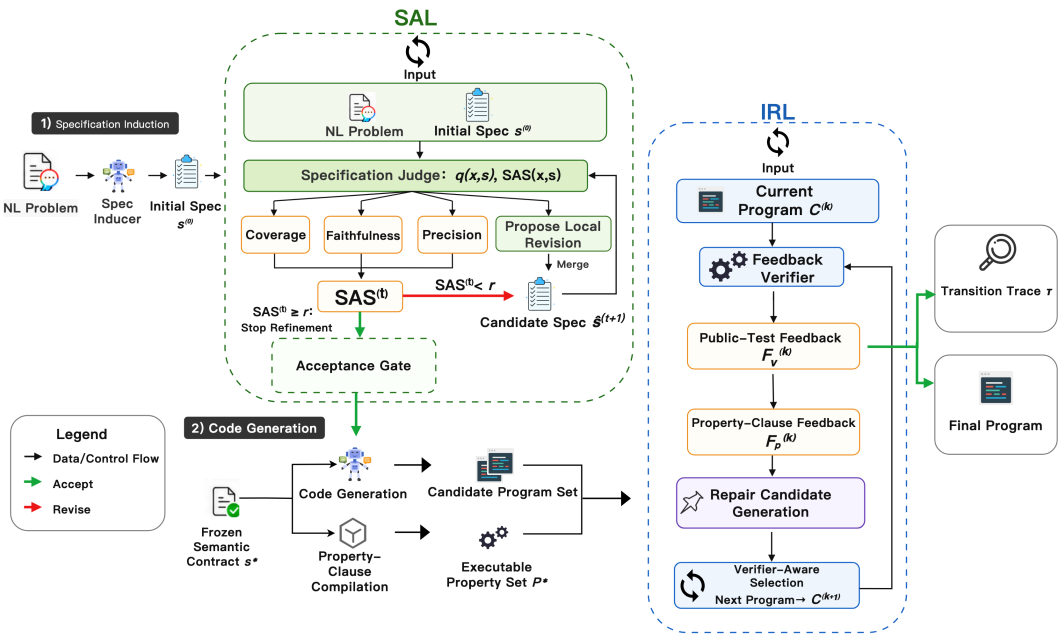

Figure 1: Overview of the semantic-contract-based dual-loop framework.

vations motivate an explicit semantic state that remains separate from executable implementation feedback.

3 METHOD

The method is motivated by a state-observability problem. An incorrect program may follow from an inadequate task interpretation, an incorrect implementation of an adequate interpretation, or both. Direct NL-to-Code exposes only the final program and cannot show where the failure entered. We therefore place an explicit semantic state between problem x and program C, revise that state before implementation, and hold it fixed during program repair.

3.1 OVERVIEW

Figure 1 divides generation into two connected stages. SAL induces an initial specification $s^{(0)}$. At semantic iteration t, it proposes a field-level patch $\Delta^{(t)}$, merges the patch with the current specification to obtain $\hat{s}^{(t+1)}$, and accepts the candidate only if it satisfies both safety and improvement conditions. An accepted candidate directly replaces the current state; a rejected candidate leaves $s^{(t)}$ unchanged and stops SAL. The current specification at termination is frozen as semantic contract $s^{\star}$, from which executable property set $P^{\star}$ is compiled.

IRL then generates an initial candidate set $\mathcal{C}^{(0)}$ from $(x, s^{\star})$ and uses public feedback to select the unique initial state $C^{(0)}$. At round k, repairs are generated only from the current program $C^{(k)}$; the resulting candidates are evaluated on public tests to select the next state. The resulting repair candidates form the round-specific set $\mathcal{C}^{(k+1)}$, from which the system selects $C^{(k+1)}$. Unselected candidates from earlier rounds do not carry over. The complete process records semantic states, implementation states, and their evidence in trace τ:

$$\begin{aligned} x \to s^{(0)} \to (q^{(0)}, \mathrm{SAS}^{(0)}) \to \cdots \to (\Delta^{(t)}, \hat{s}^{(t+1)}) \\ \to \mathrm{Accept}^{(t+1)} \to s^{(t+1)} \to \cdots \to s^{\star} \to P^{\star}, \\ (x, s^{\star}) \to \mathcal{C}^{(0)} \to C^{(0)} \to \cdots \to C^{(k)} \\ \to (F_v^{(k)}, F_p^{(k)}) \to \mathcal{C}^{(k+1)} \to C^{(k+1)} \to \cdots \to \tau. \end{aligned} \tag{1}$$

This boundary does not treat $s^\star$ as a proof of task intent. The contract is an LLM-induced, inspectable target; SAS is a directional evaluation signal; and only a limited subset of obligations can be compiled into executable clauses. Algorithm 1 in Appendix A gives the complete control flow.

3.2 Semantic Alignment Loop

SAL contains a specification inducer, a prompt-driven semantic judge, a field-level revision module, and an acceptance gate. The six central field groups correspond to recurrent failure surfaces identified in prior defect and specification-misunderstanding studies (Tambon et al., 2025; Wang et al., 2025; Tian et al., 2025; Wu et al., 2025): task summary s_{sum}, I/O protocol s_{io}, constraints s_{cons}, rules s_{rules}, edge cases s_{edge}, and candidate properties s_{prop}:

$$s \triangleq (s_{\text{sum}}, s_{\text{io}}, s_{\text{cons}}, s_{\text{rules}}, s_{\text{edge}}, s_{\text{prop}}). \tag{2}$$

The decomposition lets the system localize a protocol error, missing constraint, unsupported rule, or boundary omission without regenerating the whole interpretation. Candidate properties are obligation statements, not trusted assertions; only clauses matching supported and reliably evaluable schemas can later become executable feedback.

The semantic judge uses the same backbone at temperature zero with a dedicated rubric and JSON output schema. Given x and s, it returns coverage, faithfulness, and precision profile $q(x, s)$ together with aggregate SAS:

$$\begin{aligned}(q(x,s), \text{SAS}(x,s)) &= \text{Judge}_{\mathbf{w}}(x,s),\\ \mathbf{w} &= (0.4, 0.4, 0.2).\end{aligned} \tag{3}$$

Coverage measures whether important obligations and edge cases are retained; faithfulness penalizes unsupported assumptions; and precision measures whether obligations are operational enough to guide generation and repair. Outputs are integers in $[0, 100]$. The pipeline validates and records the aggregate returned under the rubric rather than recomputing a post-hoc dot product, and it does not treat SAS as an independent ground-truth label. Coverage and faithfulness receive equal primary weight because either omission or invention can redirect generation; precision receives lower weight because operational detail cannot compensate for missing or distorted semantics. Appendix B.3 reports the sensitivity experiment supporting this fixed setting.

The judge additionally returns grounded issue evidence, issue types, at most three target fields, protected fields, and a field-level edit plan. A patch may change only target fields, may not change protected fields, and must produce a material semantic difference. A candidate outside these constraints is rejected before re-evaluation.

The acceptance gate separates non-regression from grounded progress:

$$\text{Accept}^{(t+1)} = 1 \iff \text{Safe}^{(t+1)} \wedge \text{Improve}^{(t+1)}. \tag{4}$$

Safe requires a valid material patch, non-decreasing SAS and faithfulness, no increase in unsupported issues, and bounded degradation in missing-issue and confidence signals. Improve requires at least one grounded gain, such as an SAS increase of at least δ, reduced issue pressure, improved precision without coverage loss, or fewer missing or blocking obligations. Appendix A gives the complete predicates. If equation 4 holds, $s^{(t+1)} = \hat{s}^{(t+1)}$; otherwise $s^{(t+1)} = s^{(t)}$ and SAL stops. The main configuration uses $\delta = 1$. Refinement is attempted only when $\text{SAS}^{(t)} < r$, the judge reports a grounded material issue, and a field-specific plan exists. We use $r = 90$, at most three attempts, and stop after one rejected or skipped revision.

No additional specification candidates are generated or ranked after this trajectory. The current specification at SAL termination is directly frozen as $s^\star$. A conservative compiler then extracts the executable subset

$$P^\star = \bigcup_{\ell \in \text{Fields}(s^\star)} \text{Compile}(\ell). \tag{5}$$

Supported schemas cover ordering, multiset preservation, output length, Boolean protocols, numeric and non-negative output, and modulo ranges. General optimality and algorithmic-correctness claims remain textual. A recognized clause may still abstain if observed and expected values do not support a reliable check.

3.3 Implementation Repair Loop

The contract-conditioned generator receives both x and $s^\star$, produces at most N_0 initial candidates, and forms $\mathcal{C}^{(0)}$. Each candidate is evaluated on public feedback tests. A fixed lexicographic rank considers complete public pass, pass ratio and count, failure progress, error type, weighted property violations, empty output, violated specification items, output-length difference, and matching lines. Private tests never participate. The selected program $C^{(0)}$ is the unique initial IRL state; unselected initial candidates are not repaired later.

At round k, the feedback verifier executes current program $C^{(k)}$ on public tests and returns structured evidence $F_v^{(k)}$, including pass status, error type, failing input, observed output, available expected output, and partial progress. SAS evaluates semantic state before implementation; $F_v^{(k)}$ evaluates executable behavior. SAL, initial selection, and IRL never query private tests. The final evaluator uses the complete public-and-private suite only after a single final program has been selected.

Let $P^\star = \{\phi_i\}_{i=1}^{N_p}$, where N_p is the number of successfully compiled clauses that can be evaluated reliably. The property oracle evaluates those clauses using the problem, current program, and verifier evidence:

$$\begin{aligned} F_v^{(k)} &= \mathrm{Verify}_{fb}(x, C^{(k)}), \\ F_p^{(k)} &= \mathrm{EvaluateClauses}(P^\star, x, C^{(k)}, F_v^{(k)}). \end{aligned} \tag{6}$$

$F_v^{(k)}$ reports concrete execution behavior, whereas $F_p^{(k)}$ records violated executable obligations and their evidence. Satisfied clauses and clauses that cannot be judged reliably produce no violation feedback. Neither feedback source may modify $s^\star$.

The repair module then generates a round-specific candidate set from the unique current program:

$$\begin{aligned} C_i'^{(k+1)} &= \mathrm{Repair}_i(x, s^\star, C^{(k)}, F_v^{(k)}, F_p^{(k)}), \quad 1 \le i \le N_r, \\ \mathcal{C}^{(k+1)} &= \{C_i'^{(k+1)} \mid 1 \le i \le N_r\}, \\ C^{(k+1)} &= \mathrm{SelectByFeedback}(\mathcal{C}^{(k+1)}, x, s^\star, P^\star). \end{aligned} \tag{7}$$

Selection uses the same public-feedback ranking as initial selection. Earlier candidate sets and their unselected members do not accumulate. If the selected result is unparsable or identical to $C^{(k)}$, no program-state change is recorded. IRL stops when the current program passes public feedback or the repair budget is exhausted:

$$\mathrm{Stop}_{\mathrm{IRL}}^{(k)} = 1 \iff \mathrm{Pass}(F_v^{(k)}) \vee (k = T_r), \qquad 0 \le k \le T_r. \tag{8}$$

The final program is then evaluated on held-out private tests. Trace labels summarize visible semantic-side, implementation-side, or mixed evidence; they support inspection but are not causal diagnoses, and the conservative policy returns `unknown` when evidence remains mixed.

4 Experiments

The evaluation is organized around four research questions. RQ1 asks whether the full semantic-contract pipeline improves end-to-end Final Pass Rate. RQ2 asks whether the effect persists beyond the main Qwen-on-LiveCodeBench setting. RQ3 asks which components account for the gain and what role the accepted contract plays. RQ4 asks what semantic and implementation evidence the trace exposes. Together, the questions progress from end-to-end correctness to robustness, mechanism, and trace evidence.

Unless noted otherwise, experiments use all 1,055 problems in LiveCodeBench release_v6 (Jain et al., 2025) and Qwen2.5-Coder-7B-Instruct (Hui et al., 2024). Public tests are available to code-candidate selection and IRL. Private tests are inaccessible to specification induction, SAL, code selection, and repair; they are queried only after a single final program has been selected. Final Pass Rate is the fraction of final programs that pass every public and private test. The solved trace category uses this same complete-suite criterion, so a public-feedback pass is not automatically counted as solved.

Method	Final Pass ↑	Δ vs. Direct ↑
Direct NL-to-Code	0.2882	–
Decomposition Only	0.3175	+0.0294
Self-Refine-style	0.3223	+0.0341
Reflexion-style	0.3365	+0.0483
SpecFix-BM (Jia et al., 2025)	0.3403	+0.0521
LPW-adapted (Lei et al., 2025)	0.3308	+0.0426
Full Dual-Loop	**0.3801**	**+0.0919**

Table 1: Main comparison on all 1,055 LiveCodeBench release_v6 problems. Final Pass is the fraction of final programs that pass the complete public-and-private suite, and ↑ marks higher-is-better. Each Δ is computed from the unrounded pass rates, so it can differ by 0.0001 from the difference of the rounded values shown. External methods are shared-protocol adaptations, not reproductions.

SAL and IRL are capped at three iterations. SAL uses threshold 90, minimum improvement $\delta = 1$, precision floor 85, and at most one rejected or skipped refinement before stopping. Initial generation uses at most two candidates and each repair round uses at most three. Self-Refine-style and Reflexion-style are controlled implementations under the same backbone, data, verifier, and repair limit rather than line-by-line reproductions of the original systems (Madaan et al., 2023; Shinn et al., 2023). The resulting values are point estimates over the fixed release; private tests are never used for model selection.

We additionally adapt two recent external systems to the same model, release, feedback restriction, and final evaluator. SpecFix-BM retains behavior clustering with four probe programs, at most four probe inputs, and one requirement-repair step (Jia et al., 2025). LPW-adapted retains plan generation, verification on at most three public examples, fixed-plan code generation, and at most three repair rounds (Lei et al., 2025). Both cap primary program candidates at 11. They are shared-protocol adaptations rather than exact reproductions; auxiliary planning, probing, and analysis calls remain method-specific, so the matched quantity is the primary program-candidate budget rather than total tokens or LLM calls.

4.1 RQ1: Does the full semantic-contract pipeline improve end-to-end code generation?

Table 1 reports the full-release comparison. Full Dual-Loop reaches 0.3801, corresponding to 401 complete-suite passes. Direct generation reaches 0.2882, while the three controlled internal variants range from 0.3175 to 0.3365. Under the shared protocol, SpecFix-BM reaches 0.3403 (359 passes) and LPW-adapted reaches 0.3308 (349 passes). Full therefore gains 9.19 percentage points over direct generation and 3.98 points over the strongest non-Full row, SpecFix-BM. Because private tests are held out until final program selection, these gains cannot be selected on the final evaluation partition. The external rows preserve the cited methods' central workflows but not their original models, data partitions, or total call budgets; they are protocol adaptations, not exact reproductions or a numerical state-of-the-art claim.

Answer to RQ1. The frozen full configuration improves over the direct, controlled, and adapted external baselines at full-release scale, supporting the complete semantic-contract and repair pipeline. RQ3 separately examines where visible movement occurs.

4.2 RQ2: Does the result generalize beyond the main backbone and benchmark?

Table 2 condenses two cross-setting checks. First, 50-problem LiveCodeBench slices use Qwen, Llama3.1-8B-Instruct (Dubey et al., 2024), and DeepSeek-Coder-6.7B-Instruct (Guo et al., 2024). Full matches the best controlled iterative baseline for Qwen and Llama and improves from 0.42 to 0.50 for DeepSeek-Coder. The gain varies by backbone, so these small slices provide consistency evidence rather than a broad model-generalization claim.

Setting (n)	Direct	Best Iter.	Full ↑
LCB–Qwen (50)	0.56	0.70	0.70
LCB–Llama (50)	0.26	0.28	0.28
LCB–DeepSeek (50)	0.30	0.42	**0.50**
HumanEval+–Qwen (162)	0.8457	0.8889	**0.9136**

Table 2: Condensed cross-backbone and external validation, reporting Final Pass rates; ↑ marks higher-is-better. Best Iter. is the strongest controlled iterative baseline in each setting.

Second, the HumanEval+ converter retains a task only when at least one executable test remains after input serialization; 162 of 164 tasks satisfy this criterion (Liu et al., 2023). Direct generation already reaches 0.8457 on this more saturated benchmark, while the best controlled iterative variant reaches 0.8889 and Full reaches 0.9136. These studies are supporting checks rather than replacements for the full release: the backbone slices are small, and HumanEval+ leaves less headroom than constraint-heavy LiveCodeBench.

Answer to RQ2. The effect is not tied to one backbone or benchmark, but its magnitude is model-dependent. The backbone slices support preliminary robustness, HumanEval+ provides an external check, and full-release LiveCodeBench remains the primary evidence.

4.3 RQ3: Which components account for the gain?

(a) Component ablation ($n = 50$)

Method	SAL	IRL	Pass ↑
Decomposition	no	no	0.66
SAL-only	yes	no	0.64
IRL-only	no	yes	0.64
Full	yes	yes	**0.72**

(b) Requirement-heavy subsets

Subset	Direct	Reflexion	Full ↑
Constraint (337)	0.1602	0.1869	**0.2315**
Format (836)	0.2512	0.2943	**0.3481**
Edge case (389)	0.3239	0.3702	**0.4216**
Semantic (526)	0.2281	0.2643	**0.3213**

Table 3: Component and requirement-focused analyses. (a) A fixed 50-problem Qwen slice with three candidates per problem, distinct from the two-candidate backbone slices in Table 2. (b) Overlapping metadata-defined subsets of the full LiveCodeBench release_v6 evaluation. Values are Final Pass rates.

Table 3(a) isolates the loops: SAL-only and IRL-only both reach 0.64, below Decomposition at 0.66, while Full reaches 0.72. Thus the slice supports the composition, not an isolated gain from either loop. Table 3(b) provides full-release evidence where semantic obligations should matter most: Full exceeds Direct by 7.13–9.77 points and Reflexion-style repair in every subset.

The transition records clarify the loops' different roles without establishing causality. Across the full release, average SAS moves from 80.15 to 82.29; SAL records 165 artifact-changing patches, while IRL records 49 passes and 36 further verifier improvements. In a historical non-compute-matched diagnostic, reopening SAL after unresolved execution failures accepted 40 revisions but only one later passed, motivating a fixed contract during repair.

Answer to RQ3. Neither loop alone accounts for the gain: their composition adds eight points on the controlled slice, and Full remains strongest on requirement-heavy subsets. The frozen contract connects SAL's semantic-state changes to IRL's verifier-guided repair.

4.4 RQ4: What semantic and implementation evidence does the trace expose?

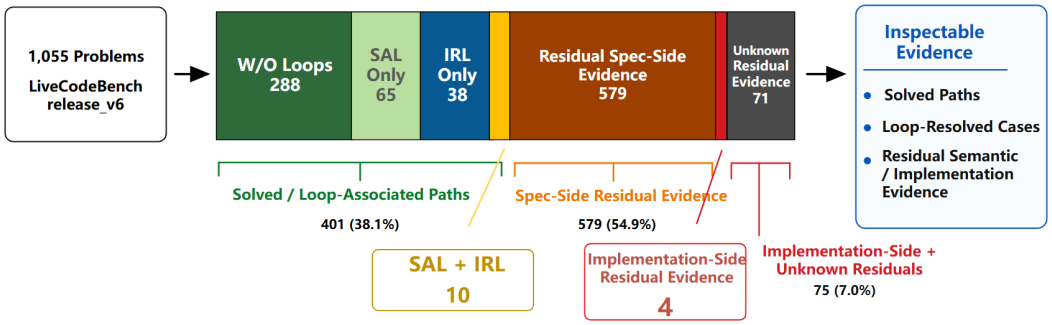

Figure 2: Final-aligned trace evidence on 1,055 LiveCodeBench release_v6 problems.

Figure 2 aligns traces with final outcomes. The 401 solved programs comprise 288 paths without loop-associated change, 65 with SAL-associated change, 38 with IRL-associated change, and 10 with both. The remaining traces contain 579 specification-side labels, 4 implementation-side labels, and 71 unknowns. A balanced 36-sample audit obtains 58.3% strict and 61.1% relaxed agreement; Appendix C gives the detailed calibration.

Figure 3 follows one trace. The task asks for the minimum partition of a binary string into powers of five without leading zeros. Initial $s^{(0)}$ already records the power-of-five condition, the no-leading-zero rule, the minimum objective, and the -1 fallback. The judge returns $q^{(0)} = (85, 80, 75)$ and $\mathrm{SAS}^{(0)} = 80$ (the rubric aggregate, not a recomputed weighted sum) and proposes a local clarification. The merged candidate yields no gain, so $\mathrm{Improve}^{(1)} = 0$, the gate rejects it, and SAL freezes $s^\star = s^{(0)}$ without generating or selecting another specification variant.

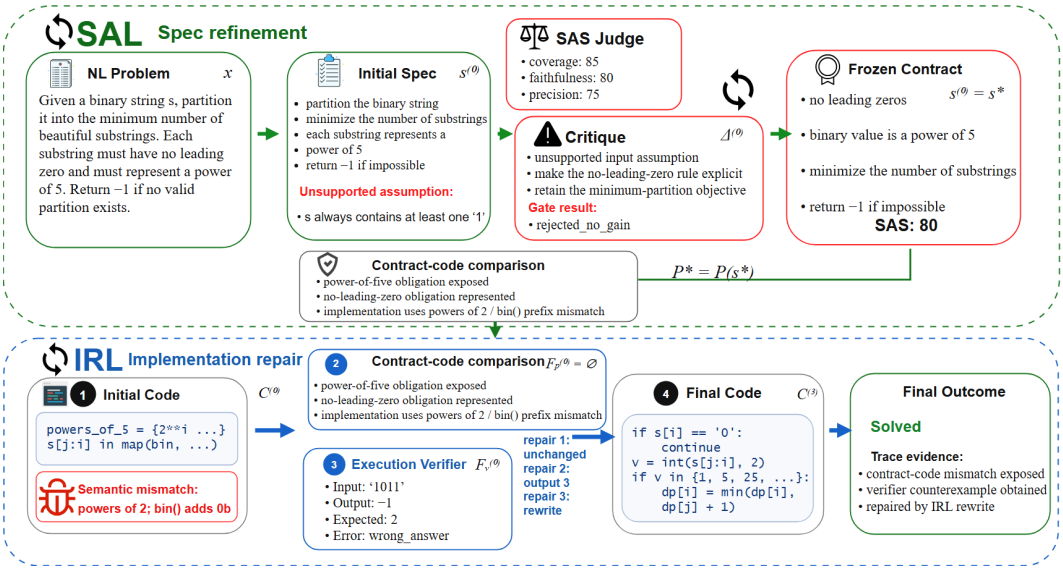

Figure 3: Case-study trace for LiveCodeBench problem 2883. The bridge $P^\star = P(s^\star)$ marks the compiled subset; the contract–code comparison boxes are explanatory, not feedback variables.

On the implementation side, $C^{(0)}$ builds powers with `2**i` and compares substrings against `bin()` strings that retain the `0b` prefix. Feedback $F_v^{(0)}$ exposes the mismatch: on `"1011"` the program returns -1, whereas `"101"` | `"1"` gives 2. IRL then parses substrings with `int(substring,2)`,

checks power-of-five membership, enforces the no-leading-zero rule, and updates the dynamic program. Because $F_p^{(0)} = \emptyset$, the trace credits this transition to verifier-guided repair: the contract makes the obligation legible, and F_v drives the correction.

Answer to RQ4. The trace exposes semantic and implementation evidence absent from direct generation: in aggregate it locates where state changed, and in a single case it shows which feedback produced a repair. It supports rather than replaces human analysis.

Together, the results support observable, reusable semantic contracts: SAL exposes semantic-state movement and IRL records verifier-level progress.

5 Conclusion

This paper presents a dual-loop NL-to-code framework that makes task semantics explicit before implementation and keeps the accepted contract fixed during verifier-guided repair. On 1,055 LiveCodeBench release_v6 problems it reaches 0.3801 Final Pass Rate, 3.98 points above the strongest shared-protocol external baseline, while keeping semantic and implementation transitions separately inspectable. The main limits are that the contract is LLM-induced rather than verified, that SAS is a directional signal produced by the same backbone, that trace labels agree with human review only 58.3% of the time, and that the reported rates are single-run point estimates. Future work will study independent specification review, confidence-aware attribution, broader settings, and budget-normalized activation.

Ethics Statement

This work evaluates program synthesis on public competitive-programming benchmarks and involves no human subjects, personal data, or deployment to end users. The only human annotation is the authors' own audit of 36 traces. The authors manually reviewed and executed the generated programs for evaluation. The system reports inspectable evidence, not verified correctness: a frozen contract and a satisfied property clause constrain an implementation without proving it correct, and presenting either as a guarantee would be a misuse of the method.

Reproducibility Statement

The Experiments section describes the benchmark partitions, feedback restrictions, model, candidate budgets, stopping conditions, and evaluation metric. The appendices provide the pseudocode, complete acceptance predicates, controlled-method definitions, sensitivity results, and supporting probes. Anonymous supplementary code contains the core SAL and IRL procedures, evaluation scripts, configuration fields, and commands needed to reproduce the reported protocol. Private tests are queried only after final program selection.

AI Use Statement

Generative AI tools were used to assist with literature discovery, language and formatting checks, and limited manuscript polishing and code editing. All cited sources, implementations, experimental results, and final claims were independently reviewed and verified by the authors, who take full responsibility for the paper.

References

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/forum?id=ktrw68Cmu9c.

Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, and Caishuang Huang. StepCoder: Improve code generation with reinforcement learning from com-

piler feedback. *arXiv preprint arXiv:2402.01391*, 2024. URL https://arxiv.org/abs/2402.01391.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. URL https://arxiv.org/abs/2407.21783.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. DeepSeek-Coder: When the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024. URL https://arxiv.org/abs/2401.14196.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, et al. Qwen2.5-Coder technical report. *arXiv preprint arXiv:2409.12186*, 2024. URL https://arxiv.org/abs/2409.12186.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md. Rizwan Parvez. MapCoder: Multi-agent code generation for competitive problem solving. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pp. 4912–4944. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.ACL-LONG.269. URL https://doi.org/10.18653/v1/2024.acl-long.269.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=chfJJYC3iL.

Haoxiang Jia, Robbie Morris, He Ye, Federica Sarro, and Sergey Mechtaev. Automated repair of ambiguous problem descriptions for LLM-based code generation. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*, pp. 367–379. IEEE, 2025. doi: 10.1109/ASE63991.2025.00038. URL https://doi.org/10.1109/ASE63991.2025.00038.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models. *ACM Trans. Softw. Eng. Methodol.*, 33(7):182:1–182:30, 2024. doi: 10.1145/3672456. URL https://doi.org/10.1145/3672456.

Haolin Jin and Huaming Chen. Uncovering systematic failures of LLMs in verifying code against natural language specifications. In *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16-20, 2025*, pp. 3819–3823. IEEE, 2025. doi: 10.1109/ASE63991.2025.00323. URL https://doi.org/10.1109/ASE63991.2025.00323.

Chao Lei, Yanchuan Chang, Nir Lipovetzky, and Krista A. Ehinger. Planning-driven programming: A large language model programming workflow. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pp. 12647–12684. Association for Computational Linguistics, 2025. doi: 10.18653/V1/2025.ACL-LONG.621. URL https://doi.org/10.18653/v1/2025.acl-long.621.

Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. AceCoder: An effective prompting technique specialized in code generation. *ACM Trans. Softw. Eng. Methodol.*, 33(8):204:1–204:26, 2024. doi: 10.1145/3675395. URL https://doi.org/10.1145/3675395.

Jia Li, Ruiqi Bai, Yangkang Luo, Yiran Zhang, Wentao Yang, Zeyu Sun, Tiankuo Zhao, Dongming Jin, Lei Li, and Zhi Jin. Bridging the gap between user intent and LLM: A requirement alignment approach for code generation. *arXiv preprint arXiv:2604.16198*, 2026. doi: 10.48550/arXiv.2604.16198. URL https://arxiv.org/abs/2604.16198.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*, 2023. URL https://arxiv.org/abs/2305.01210.

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. Self-Refine: Iterative refinement with self-feedback. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.

Mayank Mishra, Prince Kumar, Riyaz A. Bhat, Rudra Murthy V, Danish Contractor, and Srikanth Tamilselvam. Prompting with pseudo-code instructions. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pp. 15178–15197. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.EMNLP-MAIN.939. URL https://doi.org/10.18653/v1/2023.emnlp-main.939.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with AlphaCodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*, 2024. URL https://arxiv.org/abs/2401.08500.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Chi-en Amy Tai, Pengyu Nie, Lukasz Golab, and Alexander Wong. NL in the middle: Code translation with LLMs and intermediate representations. *arXiv preprint arXiv:2507.08627*, 2025. URL https://arxiv.org/abs/2507.08627.

Florian Tambon, Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Giuliano Antoniol. Bugs in large language models generated code: an empirical study. *Empir. Softw. Eng.*, 30(3):65, 2025. doi: 10.1007/S10664-025-10614-4. URL https://doi.org/10.1007/s10664-025-10614-4.

Zhao Tian, Junjie Chen, and Xiangyu Zhang. MuFix: Fixing large language models' specification misunderstanding for better code generation. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*, pp. 1514–1526. IEEE, 2025. doi: 10.1109/ICSE55347.2025.00108. URL https://doi.org/10.1109/ICSE55347.2025.00108.

Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. Compilable neural code generation with compiler feedback. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (eds.), *Findings of the Association for Computational Linguistics: ACL 2022, Dublin, Ireland, May 22-27, 2022*, volume ACL 2022 of *Findings of ACL*, pp. 9–19. Association for Computational Linguistics, 2022. doi: 10.18653/V1/2022.FINDINGS-ACL.2. URL https://doi.org/10.18653/v1/2022.findings-acl.2.

Zhijie Wang, Zijie Zhou, Da Song, Yuheng Huang, Shengmai Chen, Lei Ma, and Tianyi Zhang. Towards understanding the characteristics of code generation errors made by large language models. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*, pp. 2587–2599. IEEE, 2025. doi: 10.1109/ICSE55347.2025.00180. URL https://doi.org/10.1109/ICSE55347.2025.00180.

Jiarong Wu, Songqiang Chen, Jialun Cao, Hau Ching Lo, and Shing-Chi Cheung. Isolating language-coding from problem-solving: Benchmarking LLMs with PseudoEval. *arXiv preprint arXiv:2502.19149*, 2025. URL https://arxiv.org/abs/2502.19149.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. OpenCodeInterpreter: Integrating code generation with execution and refinement. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 12834–12859. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.FINDINGS-ACL.762. URL https://doi.org/10.18653/v1/2024.findings-acl.762.

A Complete Acceptance Predicates and Pseudocode

Let $u^{(t)}$, $m^{(t)}$, and $b^{(t)}$ be the numbers of unsupported, missing, and blocking issues returned by the judge; let $c^{(t)}$ be judge confidence, $d^{(t)}$ the number of material semantic items, and $\rho^{(t)}$ aggregate issue pressure. Hatted quantities refer to candidate $\hat{s}^{(t+1)}$. $\mathrm{Valid}^{(t+1)}$ requires a parseable candidate and score record, a patch confined to target fields and disjoint from protected fields, and at least one material semantic change. The complete non-regression predicate is

$$\begin{aligned}\mathrm{Safe}^{(t+1)} \iff \mathrm{Valid}^{(t+1)} \wedge \mathrm{SAS}(x, \hat{s}^{(t+1)}) \geq \mathrm{SAS}^{(t)} \wedge \hat{q}_{\mathrm{faith}}^{(t+1)} \geq q_{\mathrm{faith}}^{(t)} \wedge \hat{u}^{(t+1)} \leq u^{(t)} \\ \wedge \left(\hat{m}^{(t+1)} \leq m^{(t)} \vee \hat{q}_{\mathrm{cov}}^{(t+1)} > q_{\mathrm{cov}}^{(t)}\right) \\ \wedge \left(c^{(t)} = 0 \vee \hat{c}^{(t+1)} = 0 \vee \hat{c}^{(t+1)} \geq c^{(t)} - 10\right).\end{aligned} \tag{9}$$

The grounded-progress predicate is

$$\begin{aligned}\mathrm{Improve}^{(t+1)} \iff \left(\mathrm{SAS}(x, \hat{s}^{(t+1)}) - \mathrm{SAS}^{(t)} \geq \delta \wedge \rho^{(t)} > 0\right) \\ \vee \left(\hat{u}^{(t+1)} < u^{(t)}\right) \vee \left(\hat{\rho}^{(t+1)} < \rho^{(t)}\right) \\ \vee \left(\hat{q}_{\mathrm{prec}}^{(t+1)} > q_{\mathrm{prec}}^{(t)} \wedge \hat{q}_{\mathrm{cov}}^{(t+1)} \geq q_{\mathrm{cov}}^{(t)} \wedge \rho^{(t)} > 0\right) \\ \vee \left(\hat{m}^{(t+1)} < m^{(t)} \wedge \hat{q}_{\mathrm{cov}}^{(t+1)} \geq q_{\mathrm{cov}}^{(t)}\right) \\ \vee \left(\hat{b}^{(t+1)} < b^{(t)} \wedge \hat{q}_{\mathrm{prec}}^{(t+1)} \geq q_{\mathrm{prec}}^{(t)}\right) \\ \vee \left(\hat{d}^{(t+1)} > d^{(t)} \wedge \hat{q}_{\mathrm{prec}}^{(t+1)} > q_{\mathrm{prec}}^{(t)} \wedge \rho^{(t)} > 0\right).\end{aligned} \tag{10}$$

Safety prevents a local revision from damaging already expressed obligations; improvement requires at least one grounded gain. The update in equation 4 commits the candidate only when both predicates hold.

Algorithm 1 expands the transition sequence in equation 1 into the complete control flow, including every record appended to trace τ.

B Additional Experimental Details

B.1 Compared configurations

The shared evaluation pipeline compares nine configurations. Direct NL-to-Code generates one program from x without iterative feedback, and Decomposition Only induces $s^{(0)}$ and generates code with neither loop. Self-Refine-style executes public tests and iteratively revises code from that feedback (Madaan et al., 2023), while Reflexion-style accumulates the same feedback as verbal reflection and repairs from that memory (Shinn et al., 2023). SAL-only induces and gates revisions and then freezes the current specification without running IRL; IRL-only generates and repairs under an unrefined $s^{(0)}$. Of the external adaptations, SpecFix-BM clusters behavior from four probe programs and performs at most one requirement repair (Jia et al., 2025), and LPW-adapted verifies and revises a plan on public examples and then holds it fixed during generation and repair (Lei et al., 2025); both stay within an 11-program cap. Full Dual-Loop freezes $s^{\star}$, selects $C^{(0)}$, and performs contract-conditioned IRL. Style-matched baselines preserve the feedback form of the cited methods, while the two external adaptations preserve their central workflows under the common model, benchmark loader, verifier, and final evaluator.

Algorithm 1 Semantic-contract-based dual-loop generation

Require: problem x; SAL budget T_s; IRL budget T_r; candidate budgets N_0, N_r
Ensure: final program C and trace τ

Stage 1: semantic alignment; initialize trace τ
$s^{(0)} \leftarrow \mathrm{InduceSpec}(x)$
$(q^{(0)}, \mathrm{SAS}^{(0)}) \leftarrow \mathrm{Judge}_{\mathbf{w}}(x, s^{(0)})$
append initial specification and judge record to τ
for $t = 0, \ldots, T_s - 1$ **do**
  if $\neg\mathrm{ShouldRefine}(s^{(t)}, q^{(t)}, \mathrm{SAS}^{(t)}, r)$ **then**
    append SAL stopping reason to τ; **break**
  end if
  $\Delta^{(t)} \leftarrow \mathrm{ProposePatch}(x, s^{(t)}, q^{(t)})$
  $\hat{s}^{(t+1)} \leftarrow \mathrm{Merge}(s^{(t)}, \Delta^{(t)})$
  $(\hat{q}^{(t+1)}, \widehat{\mathrm{SAS}}^{(t+1)}) \leftarrow \mathrm{Judge}_{\mathbf{w}}(x, \hat{s}^{(t+1)})$
  append patch, candidate, judge record, and gate decision to τ
  if $\mathrm{Safe}^{(t+1)} \wedge \mathrm{Improve}^{(t+1)}$ **then**
    $s^{(t+1)} \leftarrow \hat{s}^{(t+1)}$; $(q^{(t+1)}, \mathrm{SAS}^{(t+1)}) \leftarrow (\hat{q}^{(t+1)}, \widehat{\mathrm{SAS}}^{(t+1)})$
  else
    $s^{(t+1)} \leftarrow s^{(t)}$; append rejection stop to τ; **break**
  end if
end for
$s^\star \leftarrow$ current specification at SAL termination; freeze $s^\star$
$P^\star \leftarrow \mathrm{CompileClauses}(s^\star)$; append $(s^\star, P^\star)$ and SAL stop to τ
Stage 2: implementation and repair
$\mathcal{C}^{(0)} \leftarrow \mathrm{GenerateCandidates}(x, s^\star, N_0)$
$C^{(0)} \leftarrow \mathrm{SelectByFeedback}(\mathcal{C}^{(0)}, x, s^\star, P^\star)$
append initial candidates, selection evidence, and $C^{(0)}$ to τ
for $k = 0, \ldots, T_r$ **do**
  $F_v^{(k)} \leftarrow \mathrm{Verify}_{\mathrm{fb}}(x, C^{(k)})$
  append $(C^{(k)}, F_v^{(k)})$ to τ
  if $\mathrm{Pass}(F_v^{(k)}) \vee k = T_r$ **then**
    append IRL stopping reason to τ; **return** $(C^{(k)}, \tau)$
  end if
  $F_p^{(k)} \leftarrow \mathrm{EvaluateClauses}(P^\star, x, C^{(k)}, F_v^{(k)})$; append $F_p^{(k)}$ to τ
  for $i = 1, \ldots, N_r$ **do**
    $C_i'^{(k+1)} \leftarrow \mathrm{Repair_i}(x, s^\star, C^{(k)}, F_v^{(k)}, F_p^{(k)})$
  end for
  $\mathcal{C}^{(k+1)} \leftarrow \{C_i'^{(k+1)} \mid 1 \le i \le N_r\}$
  $C^{(k+1)} \leftarrow \mathrm{SelectByFeedback}(\mathcal{C}^{(k+1)}, x, s^\star, P^\star)$
  retain $C^{(k)}$ if the selected result is invalid or unchanged
  append repair candidates, selection evidence, and resulting state to τ
end for

B.2 Intermediate-representation probe

This 50-problem Qwen probe supports representation design rather than the headline result. Because each representation adds a generation stage and budgets are not matched, it does not show that specifications outperform direct prompting; it only identifies the strongest tested explicit intermediate for later refinement and property compilation.

B.3 SAL hyperparameter sensitivity

The sweep uses a fixed 50-problem sample with seed 2027, disables IRL, and generates one deterministic code candidate at temperature zero from the specification current when SAL terminates.

Method	Final Pass ↑
Direct NL-to-Code	**0.700**
Plan-to-Code	0.560
Pseudocode-to-Code	0.460
Spec-to-Code	0.600

Table 4: Intermediate-representation probe on 50 Qwen problems, using one program per representation.

It varies $r \in \{80, 85, 90, 95\}$ under the paper weights and compares the paper, uniform, coverage-heavy, faithfulness-heavy, and precision-heavy profiles at $r = 90$. Only public feedback is available, so the sweep isolates SAL-side sensitivity from implementation repair and multi-candidate selection.

Table 5 reports the slice. Every threshold from 80 to 95 passes the same 20 problems. Beyond the table, the final contracts at $r = 90$ and $r = 95$ are identical on all 50 problems and $r = 85$ differs on one; accepted revisions plateau at 15 from $r = 90$, which is why we use 90.

Group	Setting	Public Pass ↑	Accepted	Calls ↓
Threshold	$r = 80$	0.400	1	5.18
	$r = 85$	0.400	14	6.90
	$r = 90$	0.400	15	7.24
	$r = 95$	0.400	15	7.24
Weights	$(0.4, 0.4, 0.2)$	0.400	15	7.24
	$(0.333, 0.333, 0.334)$	0.420	13	7.18
	$(0.6, 0.2, 0.2)$	0.400	10	7.26
	$(0.2, 0.6, 0.2)$	0.400	7	7.14
	$(0.2, 0.2, 0.6)$	0.400	6	6.94

Table 5: SAL sensitivity on a fixed 50-problem public-feedback slice.

The one-problem difference between the paper and uniform weights is descriptive: their 95% Wilson intervals, $[0.276, 0.538]$ and $[0.294, 0.558]$, overlap substantially. Changing $\mathbf{w}$ also changes the SAS scale, so aggregate SAS values are not compared across profiles; the sweep supports local stability rather than a uniquely optimal configuration.

B.4 Property-clause coverage

Zero violations in Table 6 mean only that no evaluated output triggered the recorded check.

Clause type	Compiled ↑	Violations
`numeric_output`	1101	53
`modulo_output`	159	0
`non_negative`	152	4
`sorted_order`	69	22
`boolean_output`	30	0
`same_length`	12	0
`same_multiset`	6	7

Table 6: Property clauses extracted from accepted contracts. Compiled counts extracted clauses; Violations counts events over verifier failures, and a clause may be evaluated more than once.

C Trace-Label Audit

The path labels in Figure 2 identify where recorded state changed, not which loop alone caused the final outcome: a residual specification-side label records semantic evidence such as a weak accepted

specification or unresolved obligation, whereas an implementation-side label requires sufficiently clear contract and verifier evidence.

A balanced 36-sample audit evaluates this distinction (Table 7). It contains 12 solved and 12 system-labeled spec-side samples, together with 12 implementation-side samples supplemented from auxiliary ablations because only four such residual labels appear in the full final-aligned run. The reviewer examines the raw task, accepted specification, final code, latest verifier signal, and system label. Strict agreement is 58.3% (21 of 36), with one additional partial match for 61.1% relaxed agreement. The accepted specification is judged fully adequate in 25 cases and partially adequate in 7 more, yielding 88.9% combined adequacy.

Trace label	n	Agree ↑	Partial	Disagree ↓
Solved	12	12	0	0
Spec-side	12	1	1	10
Impl.-side	12	8	0	4
Overall	36	21	1	14

Table 7: Human audit of trace-based attribution labels.