Verifiers in the Loop: Decoding Time Verification for Code Translation

Tianyang Zhou
University of Illinois Urbana-Champaign
tz64@illinois.edu

Somesh Jha
University of Wisconsin–Madison and Google
jha@cs.wisc.edu

Mihai Christodorescu
Google
christodorescu@google.com

Kirill Levchenko
University of Illinois Urbana-Champaign
klevchen@illinois.edu

Varun Chandrasekaran
University of Illinois Urbana-Champaign
varunc@illinois.edu

Abstract

Test-time scaling is an important mechanism for improving large language models, especially on tasks with deterministic verifiers. Code translation is a canonical example: the source program constrains valid outputs, while compilers, type checkers, and behavioral checks provide exact pass/fail feedback. Outcome-only approaches apply these verifiers only after generation, which is inefficient because early errors corrupt the autoregressive context and are rarely corrected later. We introduce *Decoding Time Verification (DTV)*, a framework that treats structural boundaries as meta steps for verifier-guided decoding. DTV interleaves generation with verifier calls under a state-machine controller that commits verified prefixes, using structural-boundary checks and structure-aware rollback to prevent error propagation while reducing wasted tokens. We evaluate DTV on C-to-Rust and JavaScript-to-TypeScript translation. Using Qwen3-4B as the primary generator under matched token budgets, DTV improves pass rates from 72.3% to 82.0% on C-to-Rust and from 33.3% to 46.0% on JavaScript-to-TypeScript relative to matched self-refinement baselines, while using fewer tokens per case; the same trend largely transfers to Gemma-4-E4B and Qwen3-30B-A3B. In the evaluated cost-matched grid, DTV achieves a more favorable tradeoff between pass rate and generated-token cost than post-hoc verification or sampling-based scaling. These results show that verifier-guided decoding is an effective use of inference-time compute for code translation.

1 Introduction

Large language models (LLMs) have become capable general-purpose code generators, but improving performance on difficult inputs is increasingly pursued at inference time rather than solely through larger pretraining runs. A growing line of work studies *test-time scaling*: allocating additional inference compute through repeated sampling, search, self-refinement, or process-level guidance to improve the performance of a frozen pretrained model [1–6].

Most existing test-time scaling methods are designed for open-ended domains where correctness is difficult to verify exactly. Consequently, they guide generation using probabilistic signals derived from the model itself or learned scorers [7, 8], such as token likelihoods, preference models, or

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

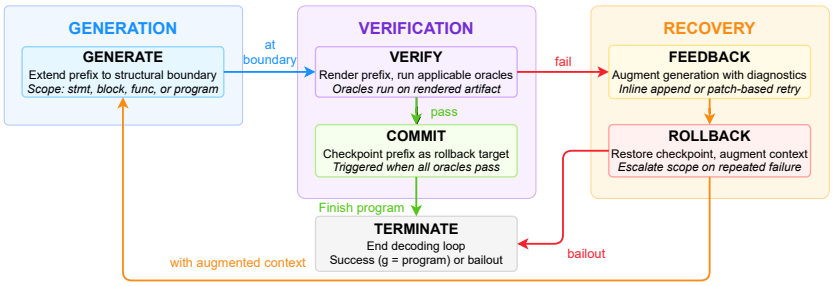


Figure 1: **GENERATION** extends the target prefix to the next structural boundary (statement, block, function, or program). **VERIFICATION** renders the prefix and runs applicable oracles, committing on pass. **RECOVERY** parses diagnostics on fail, rolls back to the last checkpoint with augmented context, and escalates scope on repeated failure. The loop terminates on program-level success or budget bailout.

intermediate reasoning heuristics. While these signals can improve reasoning quality in open-ended settings, coding tasks impose discrete semantic and syntactic correctness constraints, making probabilistic guidance only weakly aligned with true success: models may assign high probability to plausible-looking yet incorrect programs, while learned reward models for code are expensive to train and brittle across languages and libraries [9, 10]. At the same time, coding tasks expose deterministic external verifiers unavailable in most domains. Compilers, type checkers, and behavioral tests provide exact pass/fail feedback together with structured diagnostics, creating an opportunity for stronger inference-time guidance. Outcome-only approaches use these signals post-hoc to filter, rerank, or repair completed samples [11–13]. This cannot influence the trajectory of generation itself: once a verifier-detectable error enters the autoregressive prefix, later decoding proceeds under a corrupted conditioning context, and models tend to continue the trajectory rather than recover [14].

One way to overcome this limitation is to let verifier signals participate directly in the autoregressive loop, governing which prefixes are admitted as conditioning context rather than only which completed samples are retained. ROCODE [15], EG-CFG [16], and VerMCTS [17] establish this in-loop premise through backtracking, execution feedback, and partial-program search. Doing so raises three coupled questions: *when* to invoke a verifier, since verifiers require structurally complete artifacts rather than arbitrary token prefixes; *how* to roll back when verification fails, balancing context preservation against error excision; and *what* to do with diagnostics beyond retry with the same context.

Building on this in-loop premise, we introduce *Decoding Time Verification* (DTV), a decoding framework that interleaves generation from a pretrained code model with deterministic verifier calls under an explicit state-machine controller[1]. As illustrated in Figure 1, DTV generates target code up to the next structural boundary (e.g., statement, block, or function), renders the current prefix into a verifier-consumable artifact, and runs the applicable verifiers. If all checks pass, the prefix is committed; else, the controller rolls back to an earlier checkpoint, updates a feedback state using verifier diagnostics, and retries generation. Unlike prior in-loop verification systems that fix a particular combination of scope, verifier type, and search mechanism [16–18], DTV separates the controller, verification scopes, and verifier suite as configurable components. It combines three key mechanisms. First, *structural-boundary verification* couples rendering and a unanimous-pass commit condition to maintain a *legal prefix invariant*: committed prefixes can be extended into programs that pass all oracles. Second, *structure-aware rollback with escalation* rolls back to the smallest scope, escalating to larger scopes when the same diagnostic persists rather than retrying more. Third, *feedback via prompt augmentation* summarizes verifier diagnostics and injects them back into the model context so retries proceed under updated information rather than blind resampling. Together, these mechanisms prevent verifier-detectable errors from propagating through the autoregressive context by ensuring generation continues only from committed verified prefixes. While we instantiate DTV on code translation, the framework in principle applies to any task with deterministic process-level verifiers and a structural renderer, such as code generation tasks with external knowledge validators. In translation, the source and retained prefix constrain retries, while earlier decisions may first fail at later uses. Diagnostic feedback and expanding repair regions address both (§ 2.2).

We evaluate DTV on two translation settings with complementary verifier structures. *C-to-Rust* combines compiler checks with differential testing against the output in the end, while *JavaScript-to-TypeScript* relies on static type and lint verification for the type annotation task. Using Qwen3-4B [19]

[1] Source code is available at: `https://github.com/qsdrqs/dtv-translation`

as the primary generator under matched generation budgets, DTV with self-refinement improves compile-pass rate from 72.3% to 82.0% on C-to-Rust and from 33.3% to 46.0% on JavaScript-to-TypeScript, while using fewer tokens overall (§ 4.1); the same gains largely transfer to Gemma-4-E4B [20] and Qwen3-30B-A3B (Appendix K). These gains are not solely due to outer-loop scaling: even without outer-loop retry, in-loop verification alone improves one-shot pass rate by +30.7 pp and +16.7 pp over direct decoding. In the cost-matched grid, DTV also achieves a more favorable tradeoff between pass rate and generated-token cost than other test-time scaling methods, including best-of-N and the compile-feedback-based S* baseline [21]. Ablations show that all three key design components of DTV contribute, with in-loop recovery providing the largest benefit (§ 4.2). Additional mechanism analyses, runtime measurements, and strict rechecking extend this evidence (Section 4.1 and Appendix N).

2 Decoding Time Verification

As discussed in § 1, post-hoc verification applies verifier feedback only after generation is complete, surfacing errors too late to guide decoding. We instead verify *during* decoding using deterministic program verifiers (e.g., compilers, type checkers, and tests) as oracles. Verifier calls are aligned with structural boundaries such as completed statements, blocks, or functions, where verdicts become meaningful and rollback can target the scope at which an error was introduced. This forms an *inner loop* of generation that complements traditional *outer-loop* strategies (e.g., self-refinement with verifier feedback) operating only on completed programs. Outer-loop verification is recovered as a special case by invoking oracles only at full-program scope.

2.1 Overview

Decoding Time Verification (DTV) instantiates this idea as a deterministic state-machine controller. At each structural boundary, the controller selects from a fixed action set: GENERATE (extend the token prefix), VERIFY (render the prefix and run oracles), COMMIT (checkpoint the current prefix), ROLLBACK (restore an earlier checkpoint), FEEDBACK (construct a retry prompt from diagnostics), and TERMINATE (end decoding); Figure 1 shows the overall control flow. While the framework is task-agnostic, we focus on code translation because it admits an abundance of deterministic oracles.

Translation setting. Code translation aims for the generated program to exhibit the same observable behavior as the source. Given a source program S, a translation model M generates a target token sequence $Y = (y_1, \ldots, y_m)$. At each step i, M defines a conditional distribution $p_M(y_i \mid y_{<i}, S)$, where $Y_{<i} = (y_1, \ldots, y_{i-1})$ is the prefix and $Y_{\leq i}$ is its length-i extension. We treat M as a fixed pretrained black-box generator and assume access to a suite of deterministic oracles whose verdicts capture correctness of the translation result.

Key abstractions. We now formalize the three abstractions needed to state our algorithm:

1. Scope. Let $\mathcal{G}$ be a finite totally ordered set of scopes ordered by granularity, with $\mathcal{G} = \{\texttt{stmt}, \texttt{block}, \texttt{func}, \texttt{program}\}$ in our instantiations and $\texttt{stmt} \preceq \texttt{block} \preceq \texttt{func} \preceq \texttt{program}$ denoting nested enclosure (a statement inside a block, a block inside a function, and so on). Each scope $g \in \mathcal{G}$ also marks a class of *structural boundaries*: positions in the target token sequence at which an instance of g ends (e.g., a function's closing brace, a statement's semicolon). Oracles can be invoked only at these boundaries, since intermediate prefixes are typically syntactically incomplete.

2. Renderer and artifacts. Partial prefixes are typically syntactically incomplete (unclosed braces, unfinished statements), and different oracles expect different inputs (a compilable snippet for a compiler, an executable for a test runner). We therefore define a deterministic *renderer*

$$\mathsf{Render} : (S, Y_{\leq i}) \mapsto A_i \in \mathcal{A} \tag{1}$$

mapping a boundary-terminated prefix to an *artifact* an oracle can process, where $\mathcal{A}$ is the artifact space equipped with a scope map $g : \mathcal{A} \to \mathcal{G}$. Render is defined only at *structural boundaries*: prefixes ending at the closing token of some $g \in \mathcal{G}$.

3. Oracle interface. Let $\mathcal{O} = \{O_1, \ldots, O_J\}$ be the suite of verification oracles. Each oracle O_j has an *applicability predicate* $F_j : \mathcal{A} \to \{0, 1\}$ indicating whether it can be meaningfully invoked on A, and returns a deterministic verdict and diagnostics:

$$O_j(S, A) = (q_j, d_j), \quad q_j \in \{0, 1, \bot\}, \tag{2}$$

where 1, 0, $\perp$ denote pass, fail, and not applicable respectively; $q_j = \perp$ iff $F_j(A) = 0$. In our instantiations F_j is typically a scope threshold $F_j(A) = \mathbf{1}[g(A) \succeq g_j^{\min}]$, but F_j may depend on other features of A (e.g., a differential-testing oracle requires the target function to have test inputs).

Objective. A decoding strategy π interleaves token generation under M with oracle invocations on intermediate artifacts, and may rollback and regenerate. Let $A_m := \mathsf{Render}(S, Y_{\le m})$ be the terminal artifact, with terminal quality $Q(S, A_m) = 1$ iff all applicable evaluation oracles pass on A_m. Given a generation budget B_{gen} that hard-caps next-token model calls (counting tokens later discarded by rollback), DTV targets

$$\max_{\pi} \quad \mathbb{E}\big[Q(S, A_m)\big] \quad \text{s.t.} \quad N_{\text{gen}}(\pi) \le B_{\text{gen}}, \tag{3}$$

where the expectation is over randomness in sampling from p_M and any stochasticity in π. § 2.2 gives our concrete, deterministic instantiation of π.

2.2 Core Mechanisms

We target the objective Eq. (3) through three mechanisms: (i) *structural-boundary verification* (§ 2.1), which maintains the legal prefix invariant through boundary-aligned rendering and a unanimous-pass commit condition; (ii) *structure-aware rollback with escalation*, which rolls back to the smallest scope and escalates to larger scopes (e.g., from a statement to its enclosing block or function) when the same diagnostic persists; and (iii) *feedback via prompt augmentation*, which surfaces unresolved oracle failures back to the model so subsequent generation conditions on what went wrong. Algorithm 1 (Appendix G) describes these mechanisms as a single loop: at each iteration the controller extends the prefix to the next structural boundary, runs applicable oracles on the rendered artifact, and either commits (on unanimous pass) or invokes feedback-augmented rollback at a chosen scope (on failure). The inner loop ends on candidate completion, on *bailout* (repeated diagnostic-anchor visits reach the recovery limit and the current artifact is returned to the outer loop), or generation budget exhaustion. The common final gate judges the returned artifact.

Meta steps. Between structural boundaries, DTV delegates token generation to the base sampler; at each boundary it pauses to invoke oracles and act on their verdicts. We call each pause a *meta step* (indexed $k = 1, \ldots, K$): let b_k denote the prefix length at meta step k (after any rollbacks), $A_k := \mathsf{Render}(S, Y_{\le b_k})$ the artifact, and $g_k := g(A_k)$ its scope. Oracles O_j with $F_j(A_k) = 1$ are applicable at meta step k. Every committed checkpoint must satisfy the *legal prefix invariant*: the committed prefix can be extended into a program that passes all oracles, so that rollback targets are always viable starting points. Enforcement is two-part: the renderer turns the current prefix into an oracle-consumable artifact, and Algorithm 1 commits only when all oracles applicable at the current scope pass. The controller below decides which action to take at each meta step.

Controller. The policy π is instantiated as a deterministic state-machine controller. At each meta step k, it consumes an observation σ_k and emits an operation ω_k that drives the next iteration. The observation aggregates the current prefix, the latest oracle outputs $o_k = \{(q_j, d_j) : F_j(A_k) = 1\}$, the feedback state $\mathcal{F}_k$, bounded retry and visit counters, and the remaining token budget. The op $\omega_k = (a_k, \ell_k, \mu_k, \beta_k)$ chooses one of six actions $a_k \in \{\textsc{Generate}, \textsc{Verify}, \textsc{Commit}, \textsc{Rollback}, \textsc{Feedback}, \textsc{Terminate}\}$, with optional fields: a rollback scope $\ell_k \in \mathcal{G}$, a retry mode μ_k, and a bailout flag β_k. Between meta steps, token generation follows the base sampler unchanged (greedy, top-k, top-p). DTV is a modular framework with reusable controller interfaces and task/model adapters (Appendix H). Appendix G gives precise semantics for each action.

Feedback. Rollback alone amounts to resampling from the same conditional distribution: although sampling diversity (e.g., varying temperature) can yield retries that differ from earlier attempts, exploring for the specific fix without diagnostic guidance is inefficient. The feedback loop therefore has two parts.

1. Feedback state. The feedback state $\mathcal{F}_k$ is the controller's working record of oracle failures not yet resolved, with each entry holding the diagnostics needed to guide repair (error message and source anchor). It feeds both the prompt augmentation step and the rollback/bailout policy. After observing o_k, the state is updated:

$$\mathcal{F}_{k+1} = \mathsf{Update}(\mathcal{F}_k, o_k). \tag{4}$$

Update is a deterministic state update that adds new failures, drops failures resolved by passing verdicts, and bounds the state's size.

2. Prompt augmentation. We communicate $\mathcal{F}_{k+1}$ back to the model by augmenting the conditioning context. Let $\phi_{k+1} := \mathsf{Encode}(\mathcal{F}_{k+1})$ be a textual block summarizing the active diagnostics, and $\widetilde{S}_{k+1} := \mathsf{Augment}(S, \phi_{k+1})$ the augmented conditioning context. Between meta steps, $\widetilde{S}_{k+1}$ replaces S in the conditioning context, so for any prefix $Y_{<i}$ the next-token distribution shifts from $p_M(\cdot \mid Y_{<i}, S)$ to $p_M(\cdot \mid Y_{<i}, \widetilde{S}_{k+1})$. Rollback-with-feedback is therefore *not* blind resampling: decoding resumes under updated feedback while S remains fixed. Two concrete retry modes (inline continuation, patch-based repair) are described below.

Rollback, escalation, and bailout. When verification fails, the controller rolls back the prefix at a scope $\ell \in \mathcal{G}$ chosen by a fixed escalation rule. Rollback starts at statement scope; when repeated retries yield the same diagnostic, the controller escalates to the enclosing block, or to the function if no block is open. Coarser scopes expose a larger region of code in which the model can revise its generation while preserving the retained context.

For example, consider a translation that must retain `message` for the later output:

```
fn consume(_: String) {}
fn show(message: String) {
    consume(message);        // Ownership moves here.
    println!("{message}");  // E0382 is reported here.
}
```

Cloning `message` at the later `println!` fails because ownership has already moved. Changing the earlier call to `consume(message.clone())` preserves `message` for the later output. Statement-only rollback cannot modify the earlier call, motivating scope escalation.

To avoid unbounded retries on unfixable errors, DTV introduces two safeguards. *Escalation*: the controller widens the rollback scope after repeated failures at finer scopes. For example, a Rust mutability error whose root cause lies several statements upstream may require escalation to block scope so the binding region itself can be regenerated (Appendix L.4). *Bailout*: when diagnostic-anchor visits reach a fixed threshold, the controller terminates the inner loop and returns the current artifact together with outstanding diagnostics to the outer loop.

In our implementation, a streaming-parser renderer maintains checkpoints over both the token prefix and structural parsing state. After rollback, retries proceed in one of two modes: *inline continuation*, which appends diagnostics as comments and resumes decoding in the same turn, or *patch-based repair*, which requests a minimal patch in a new turn (Appendix I shows the prompt format and an example exchange for each mode). Fixed counters govern scope escalation, retry-mode changes, and bailout (Appendix H).

3 Experimental Setup

Tasks and datasets. We evaluate DTV on two complementary translation tasks:

1. *C to Rust* (unsafe-to-safe systems translation): we sample 300 C programs from CodeNet [22], with test inputs for differential testing generated via AFL++ fuzzing with LLM-based seeding; we use `rustc` as oracle and differential testing for post-hoc behavioral equivalence.
2. *JavaScript to TypeScript* (dynamic-to-static typing): we sample 150 self-bundled programs from TypeWeaver [23] that do not type-check as plain JavaScript, using `tsc` and ESLint's `@typescript-eslint/typedef` rule as oracles; executable tests are unavailable.

For comparisons with other baselines and the Gemma cross-model analysis, we use cost-matched subsets of 200 (C to Rust) and 100 (JS to TS), drawn from the same pools. Qwen3-30B-A3B and the mechanism comparisons use the first 100 cases per relevant task. Per-task DTV instantiation, sampling pipelines, and oracle filtering choices are in Appendix E.

Models. DTV requires direct control over decoding (boundary stops, rollback, feedback injection); closed-weight inference APIs do not expose this control surface, so we use open-weight models throughout. Our primary translation model is Qwen3-4B-Instruct [19]; we additionally use Gemma-4-E4B-it [20] to test generalization across model families and Qwen3-30B-A3B-Instruct to evaluate a stronger generator. DTV and the baselines share the same model and decoding configuration in every comparison; full hyperparameters are in Appendix F.

Baselines and budget. We decompose test-time decoding strategies along two orthogonal axes. The *inner loop* is either **naïve** (the outcome-only configuration of § 2.2: complete generation in one

pass with no in-loop verification) or **DTV** (in-loop verification, rollback, and feedback; § 2.2). The *outer loop* is either **one-shot** (a single attempt), **self-refine** (sequential regeneration with diagnostic feedback when available), or **best-of-N (BoN)** (N independent attempts, returning the first oracle-passing candidate). We evaluate five inner×outer combinations—naïve/one-shot, naïve/self-refine, naïve/BoN, DTV/one-shot, DTV/self-refine—plus S* [21][2] as a process-verified reference baseline combining parallel sampling with compile-driven self-debug. All self-refine configurations share the same per-case generation budget of $16\times$ source tokens, counting all output tokens within a case.

Metrics. The primary metric is the post-hoc compilation pass rate (`rustc` for C to Rust; `tsc` with strict mode disabled and ESLint zero-`typedef` for JS to TS). All methods share the unfiltered final gate; selected diagnostic filtering applies only to DTV's intermediate search (Appendix E). On C to Rust we additionally report functional success rate via differential testing to verify that compile-pass gains do not mask semantic regressions (Appendix J.7). We report the average per-case output tokens as the cost metric, supplemented by elapsed time and verifier calls. A shared `tsc --strict` recheck evaluates the saved JS-to-TS outputs under a stronger criterion (Appendix J.4).

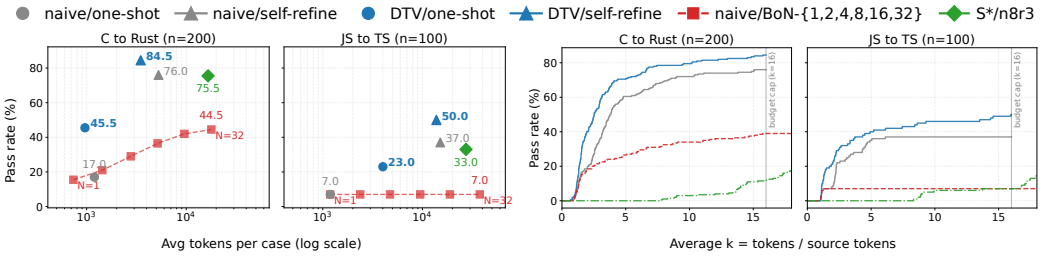
4 Results

Through our evaluation, we aim to answer the following questions:

- **RQ1.** Under equal per-case token budgets, does (i) DTV perform better than baselines on pass rate versus token cost on both tasks, and (ii) the gain transfer to different generator model families?
- **RQ2.** Do DTV's key mechanisms all contribute positively to its performance?
- **RQ3.** (i) On which cases does DTV outperform naïve, (ii) why does it do so, and (iii) how does this picture hold under multi-round self-refine?

4.1 Comparing DTV with baselines on pass rate and token efficiency

Under a fixed per-case generation budget of $16\times$ source tokens, DTV/self-refine Pareto-dominates naïve/self-refine and lies in the upper-left region of the cost-vs-pass-rate frontier on both tasks (cumulative view in Figure 2b; full-set Pareto view in Figure 6 in Appendix J.1). It reaches 82.0% on C to Rust and 46.0% on JS to TS, exceeding naïve/self-refine by +9.7 pp and +12.7 pp.



(a) Per-case token cost (log scale) vs. compile pass rate, (b) Cumulative pass rate vs. per-case token budget k comparing against naïve, best-of-N and S*

Figure 2: DTV/self-refine Pareto-dominates naïve/self-refine and sits on the upper-left of the cost-pass-rate frontier on both tasks, above the BoN-N and S* frontiers at matched per-case token cost.

DTV/self-refine is also more token-efficient than naïve/self-refine while attaining a higher pass rate: on C to Rust the two configurations average $k = 5.28$ and $k = 7.18$ tokens per case respectively.

Cumulative scaling of DTV vs. naïve. The cumulative pass-rate curves over the per-case token budget k (Figure 2b) reveal two structural differences. First, DTV exhibits an early-rise effect: its step curve begins climbing at substantially smaller k than naïve/self-refine on both tasks, because in-loop verification and repair admit successful programs well before naïve's first complete sample is generated. Second, on JS to TS, the two regimes diverge in the long-run: naïve/self-refine plateaus once additional tokens cease to yield further passes, while DTV/self-refine continues to rise. Once a

[2]Configured with independent attempts N=8, rounds R=3, and compile-error-based selection, deviating from the original defaults [21] to fit the per-case token budget and fair comparison (clarified in Appendix E).

generation has settled on an under-typed structure, further refinement attempts tend not to recover from it; in-loop type-annotation feedback, by contrast, redirects generation away from these basins.

Scaling through in-loop verification. With a single attempt and no outer-loop retry, DTV raises the pass rate from 16.3% to 47.0% on C to Rust and from 4.7% to 21.3% on JS to TS (Figure 6 in Appendix J.1). Its in-attempt rollbacks and retries cost extra tokens, but the pass rate grows $2.9\times$ and $4.6\times$ while average tokens per case (k, generated tokens per source token) grow only $1.3\times$ and $2.5\times$ (Table 5 in Appendix J.2). Adding self-refinement raises the pass rate further, to 82.0% and 46.0%, so DTV complements rather than replaces outer-loop test-time scaling.

This compile-pass improvement is not accompanied by a regression in behavioral correctness: on C to Rust, DTV/self-refine attains a full-functional success rate of 8.3% compared with 9.3% for naïve/self-refine, a difference indistinguishable from zero under the paired test (Appendix J.7). DTV's in-loop oracle on C to Rust is `rustc` alone, with differential testing reserved for post-hoc evaluation, so the controller receives no behavioral-equivalence signal during decoding; this guardrail finds no evidence of eroded behavioral correctness while optimizing compile pass.

The JS-to-TS gain is robust to a stricter final criterion: adding `tsc --strict` preserves DTV's pass-rate advantage across all evaluated models (Appendix J.4).

Cross-model robustness. The same in-loop verification mechanism transfers to a second model family. We re-evaluate DTV/self-refine against naïve/self-refine using Gemma4 as the generator on both tasks (Appendix K). DTV exceeds naïve at the budget cap on three of the four model-task pairs and reduces normalized cost on three. The exception is C to Rust on Gemma4, where naïve already reaches 93.5% at the cap; under tight budgets ($k \leq 7$), DTV/self-refine still leads naïve on this cell with a peak advantage of about $+10$ pp at $k \approx 4$, and the curves cross near $k \approx 7$ where naïve's outer-loop retry over residual hard cases yields more marginal passes than DTV's verify-and-rollback.

The benefit extends to the stronger Qwen3-30B-A3B generator: final pass rates rise from 92% to 97% (C to Rust) and 57% to 66% (JS to TS), with gains already in the one-shot setting (Appendix K.1).

Comparing with other baselines. We additionally compare DTV/self-refine against best-of-N (BoN, $N \in \{1, 2, 4, 8, 16, 32\}$) and S* at matched per-case budget; DTV/self-refine remains on the upper-left frontier against both on both tasks. On C to Rust, BoN rises from 15.5% at $N=1$ to 44.5% at $N=32$, but no N reaches DTV/self-refine's frontier: small N ($N \leq 4$) is cheaper yet caps below 29.0%; large N ($N \geq 8$) costs more yet trails by at least +40.0 pp (Figure 2a). On JS to TS, BoN stays at 7.0% for all N as cost grows, suggesting resampling repeats the same under-typed structure.

S* reaches 75.5% on C to Rust and 33.0% on JS to TS, versus DTV/self-refine's 84.5% and 50.0%, while consuming roughly $4.7\times$ DTV's per-case tokens on C to Rust and $2\times$ on JS to TS. DTV's advantage stems from verification timing: S* compiles only after each completed attempt, whereas DTV verifies at structural boundaries within a single attempt, converting the same signal into early localized rollback rather than full restart. The full baseline grid is in Appendix J.5.

Wall-clock tradeoff. DTV saves time with `rustc` but adds overhead with serial TypeScript/ESLint. From Qwen3-4B to Qwen3-30B-A3B, DTV/naïve time ratios improve from 0.79 to 0.57 and from 1.85 to 1.26, respectively. This smaller relative overhead favors DTV as generation becomes expensive relative to a fixed verifier stack. Table 6 and Appendix J.3 give timings and call counts.

Takeaway. In pass rate versus generated tokens, DTV/self-refine Pareto-dominates naïve/self-refine and lies above all baseline frontiers on both tasks, with the gain transferring across model families.

4.2 Ablation study on DTV's three key mechanisms

Table 1: Ablations on 100 C-to-Rust cases. All four configurations share the self-refine outer loop and the same per-case token budget. Δ columns report the ablation minus DTV-full.

Configuration	Pass rate	Δ pp	Avg. tokens	Δ tokens
DTV-full	85.0%	–	3580	–
DTV-no-feedback	74.0%	-11.0	4138	+558
DTV-no-escalation	79.0%	-6.0	3623	+43
DTV-detect-and-abort	70.0%	-15.0	4417	+837

We ablate each of DTV's three key mechanisms (§ 2.2) in turn, holding the remainder of DTV (including the outer self-refine loop) fixed:

- *DTV-no-feedback* (ablates feedback via prompt augmentation): rollback without injecting diagnostics during inner generation.
- *DTV-no-escalation* (ablates structure-aware rollback with escalation): always roll back to statement scope, never widening to block or function.
- *DTV-detect-and-abort* (disables in-loop recovery): terminate the inner loop on the first verification failure, deferring recovery to outer self-refine.

DTV with all three mechanisms enabled (hereafter DTV-full) attains the highest pass rate among the four configurations, and disabling any of the three reduces pass rate (Table 1). DTV-full reaches 85.0% on the first 100 C-to-Rust cases. Removing inner-loop diagnostic feedback drops pass rate to 74.0% (-11.0 pp), restricting rollback to statement scope drops it to 79.0% (-6.0 pp), and disabling structure-aware inner recovery (detect-and-abort) drops it to 70.0% (-15.0 pp).

Within an equal per-case token budget. DTV-full converts more of the budget into passing translations than the ablations: 3580 avg tokens per case versus 4138, 3623, and 4417 for the three ablations. The detect-and-abort ablation repeatedly delegates recovery to the outer self-refine loop: 13.5 outer rounds versus 3.1 for DTV-full, roughly $4\times$. DTV-full achieves this by spending more tokens on each successful translation than the ablations: 1694 per pass versus 1412, 1437, and 1173 (detailed in Appendix L.1). The extra per-success tokens fund the in-loop verification and structured-recovery work that converts otherwise-failing cases into passes.

Differential-testing pass rates. DTV-full attains 10.0%, while no-feedback, no-escalation, and detect-and-abort reach only 8.0%, 5.0%, and 8.0% respectively. DTV-full thus leads on this behavioral criterion as well. Appendix L.2 reports paired McNemar tests on this 100-case ablation cohort.

The inner-1shot pass rate. This decomposes each mechanism's effect between inner DTV and outer self-refine. Removing diagnostic feedback changes inner-1shot by -23.0 pp: without an explicit signal about what failed, the inner loop's resampling tends to reproduce the same error pattern, deferring recovery to outer self-refine. Disabling structured inner recovery (detect-and-abort) changes inner-1shot by -30.0 pp, the largest first-pass drop: every inner verification failure discards in-loop state and forces outer self-refine to retry from scratch with only post-hoc diagnostic context. Restricting rollback to statement scope leaves inner-1shot unchanged at 44.0% but limits recovery on cases where a diagnostic detected at one statement requires repair at an earlier committed statement (e.g., rustc's "cannot assign to immutable variable" detected at a later mutation whose root cause is an earlier `let` binding, which statement-scope rollback cannot reach). DTV-full's scope escalation does occasionally over-rollback and discard correctly-committed prefix in exchange; the net effect is -6.0 pp final pass rate for no-escalation relative to full. Detailed analysis is in Appendix L.4.

Takeaway. All three of DTV's mechanisms contribute, and removing any one reduces pass rate. In-loop recovery contributes the largest gain compared to diagnostic feedback and scope escalation.

4.3 Deep analysis of DTV's gain over the naïve baseline

We take DTV's gain apart through three sub-questions: (i) on which cases DTV outperforms naïve, (ii) why it does so, and (iii) whether the same picture holds under multi-round self-refine. The analysis uses the 300 C-to-Rust and 150 JS-to-TS samples from § 4.1.

What kind of error DTV helps? To localize where DTV's gain comes from in the one-shot regime, we measure DTV's rescue rate (the fraction of naïve-failed cases that DTV solves) separately for each type of fix the case requires. Lacking a ground-truth oracle for fix location, we use naïve's self-refine behavior as a proxy: when naïve's first generation (R1) fails, its second generation (R2) attempts to repair R1, and we take the lines R2 edits in R1 as an approximation of the necessary fix location. We label each case's *fix shape*, the relation between R2's edited lines and R1's flagged error lines, as LOCAL (every edit lands on a flagged error line), NONLOCAL (none does), MIXED (partial overlap), or UNKNOWN (R2 rewrites more than 50% of R1, excluded as uninformative); a more rigorous variant using naïve's final passing round R_{final} in place of R2 yields the same conclusion on a smaller set of cases (Appendix M.1). On both tasks, DTV's rescue rate decreases monotonically as the necessary fix moves further from the failing line, with the LOCAL-to-NONLOCAL gap reaching 20.4pp on C to Rust and 29.3pp on JS to TS (Table 2; full breakdown and small-bin qualification in Appendix M.2). An edit away from the diagnostic line may still lie within an enclosing block or function, which motivates scope escalation (§ 4.2); broader nonlocal restructuring remains harder.

Task	Local	Mixed	Nonlocal	$\Delta_{\text{L-N}}$
C → Rust	50.0% (12)	43.5% (168)	29.6% (27)	+**20.4** pp
JS → TS	29.3% (58)	11.9% (67)	0.0% (2*)	+**29.3** pp

Table 2: DTV/one-shot rescue rate on naïve R1-failure cases by fix shape (R1→R2 diff). Cells show rate with sample number n; color tracks rate; $*$ marks $n < 5$. Rescue rate decreases monotonically LOCAL→NONLOCAL on both tasks. Full breakdown in Table 22 in Appendix M.2.

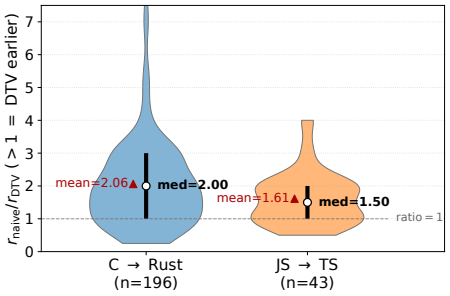


Figure 3: Paired round-count ratios ($r_{\text{naïve}}/r_{\text{DTV}}$) on the both-pass subset of self-refine. Markers: median (white circle), mean (red triangle).

Why DTV outperforms post-hoc verification? DTV and naïve differ in when they invoke verification: DTV intervenes during generation, while naïve verifies only the completed program. This timing matters because autoregressive language models condition each token's distribution on the preceding prefix [14]: an error introduced early biases subsequent tokens toward extending the error rather than recovering from it, so intervening early should reduce error accumulation compared to post-hoc verification. To check whether DTV's interventions reduce error counts, we restrict to error codes that appear in both DTV's inner-loop trace and naïve's one-shot loop, and on each rescued case compute the per-case shared-code edit-concentration ratio: distinct DTV inner-loop diagnostic emissions on the shared codes divided by distinct naïve R1 emissions on the same codes (Appendix M.4); a ratio below 1 means DTV emits fewer per-shared-code diagnostics than naïve. On JS to TS the ratio is well below 1 (median 0.50, mean 0.71, $n = 26$): a missing or wrong type annotation propagates to every downstream use site, so catching it early prevents many derivative errors. On C to Rust the ratio is near parity (median 1.00, mean 1.19, $n = 57$); the suppression effect is weaker because rustc's main error families (borrow check, lifetime, type mismatch) do not compound across use sites the way TypeScript type errors do.

How DTV saves self-refine tokens? On cases both methods eventually pass (Figure 3), DTV's earlier intervention translates into fewer self-refine rounds (median paired ratio of naïve's rounds to DTV's: 2.00 on C to Rust, 1.50 on JS to TS), and the round savings carry through to tokens on C to Rust (DTV uses fewer tokens on 67.9% of cases, median paired difference -208 tokens). On JS to TS the median still favors DTV (-61 tokens), but the mean reverses (+220 tokens), driven by a small tail of nonlocal-fix cases (e.g., typedef errors whose correct annotation depends on usage elsewhere) where the inner loop re-emits the same diagnostic and rolls back without converging. A detailed breakdown is in Appendix M.6.

Takeaway. DTV's gain concentrates on cases with local fixes while weakens on nonlocal ones, where early intervention prevents derivative errors and reduces self-refine rounds.

5 Related Work

DTV sits within test-time compute scaling, in the process-level supervision sub-family that emits step-level signals during generation [1–3, 5, 7]; its distinguishing feature is that signals come from deterministic verifiers rather than learned scorers or repeated sampling. Appendix D provides a complete discussion; Figure 4 in Appendix C contrasts DTV against the two dominant prior paradigms. We summarize the closest threads here.

Process supervision. Process reward models attach learned step-level scores and outperform outcome-only verifiers on multi-step reasoning [4, 10, 24–27]; SemGuard [28] similarly applies a learned line-level semantic evaluator during code generation and rolls back flagged lines. DTV produces step-level signals from *deterministic oracles* instead, yielding exact rather than probabilistic verdicts, removing the need for annotation or training, and providing structured diagnostics to guide repair.

Verifier-guided generation. Outcome-only methods apply verifiers to completed programs for filtering, ranking, or single-shot repair [6, 11–13, 21]. Recent work on probabilistic programs of

thought reuses token-level uncertainty to obtain cheap candidate variants before post-hoc verification [29]. In-loop methods place verifiers *inside* generation: line-level Python execution feedback [16], function-level MCTS over Dafny/Coq partial programs [17], and verifier-driven tree-search refinement on top of a self-improving training loop for Dafny-to-Verus [18]. DTV differs in three respects: adaptive structural-boundary granularity (statement to program) rather than a fixed scope, hard state-machine control rather than stochastic search, and multi-oracle composition (compiler, type checker, linter).

ROCODE [15] uses program-analysis-triggered backtracking, diagnostic-line or maximum-entropy-statement localization, and failed-path probability decay. DTV instead combines nested structural repair regions with diagnostic-conditioned regeneration, preserving unaffected context. Appendix N examines the roles of diagnostic guidance and editable repair scope. Concurrent work, Hydra [30], couples an incremental compiler checker with checkpoint rollback to reduce the cost of reaching statically valid C/C++ code, while DTV requires no verifier modification and raises pass rate under a fixed token budget.

Constrained decoding. Token-level recognizers (grammars, regexes, type automata) enforce hard syntactic constraints during sampling [31–40]. They cannot invoke semantic verifiers (full type checking, differential testing) on intermediate program scopes, since these require structurally complete artifacts. DTV is complementary: constrained decoding can enforce syntactic invariants while DTV adds semantic verification on top.

6 Conclusion

Outcome-only test-time scaling for code applies deterministic verifiers (compilers, type checkers, tests) post-hoc, after errors have compounded the whole generated code. We introduce *Decoding Time Verification* (DTV), a decoding framework that interleaves generation with verifier calls, combining structural-boundary verification, structure-aware rollback, and feedback via prompt augmentation. With Qwen3-4B, DTV/self-refine raises compile pass rate from 72.3% to 82.0% on C to Rust and from 33.3% to 46.0% on JS to TS using fewer tokens per case than naïve/self-refine, Pareto-dominating naïve/self-refine and sitting above the best-of-N and S* frontiers at matched token cost; gains transfer to Gemma-4-E4B on three of four pairs. Additional experiments with Qwen3-30B-A3B show gains on both tasks, and shared strict rechecking preserves the JS-to-TS advantage. Key challenges remain: adapting recovery decisions to the remaining budget, extending the approach beyond translation, and reducing the implementation effort for new models and tasks (Appendix A). We hope DTV motivates future work on decoding-time control with deterministic oracles.

Acknowledgments and Disclosure of Funding

We thank the anonymous reviewers and the area chair for their valuable feedback on this paper. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112590131. Approved for public release; distribution is unlimited. This work used the Delta and DeltaAI systems at the National Center for Supercomputing Applications (awards OAC 2005572 and OAC 2320345) through allocation CIS251314 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program [41], which is supported by U.S. National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296. The authors declare no competing interests.

References

[1] Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM Test-Time Compute Optimally Can be More Effective than Scaling Parameters for Reasoning. In *International Conference on Learning Representations*, 2025.

[2] Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candes, and Tatsunori Hashimoto. S1: Simple test-time scaling. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 20286–20332. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.1025.

[3] Yixin Ji, Juntao Li, Yang Xiang, Hai Ye, Kaixin Wu, Kai Yao, Jia Xu, Linjian Mo, and Min Zhang. A Survey of Test-Time Compute: From Intuitive Inference to Deliberate Reasoning, January 2025.

[4] Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's Verify Step by Step. In *International Conference on Learning Representations*, 2024.

[5] OpenAI. OpenAI o1 System Card, December 2024.

[6] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, December 2022. ISSN 0036-8075, 1095-9203. doi: 10.1126/science.abq1158.

[7] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large Language Monkeys: Scaling Inference Compute with Repeated Sampling, July 2024.

[8] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik R Narasimhan. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Advances in Neural Information Processing Systems*, 2023.

[9] Claudio Spiess, David Gros, Kunal Suresh Pai, Michael Pradel, Md Rafiqul Islam Rabin, Amin Alipour, Susmit Jha, Prem Devanbu, and Toufique Ahmed. Calibration and Correctness of Language Models for Code. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 540–552. IEEE, April 2025. doi: 10.1109/icse55347.2025.00040.

[10] Muhammad Khalifa, Rishabh Agarwal, Lajanugen Logeswaran, Jaekyeom Kim, Hao Peng, Moontae Lee, Honglak Lee, and Lu Wang. Process Reward Models That Think, April 2025.

[11] Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. LEVER: Learning to Verify Language-to-Code Generation with Execution. In *International Conference on Machine Learning*, 2023.

[12] Bei Chen, Fengji Zhang, A. Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code Generation with Generated Tests. In *International Conference on Learning Representations*, July 2022.

[13] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching Large Language Models to Self-Debug. In *International Conference on Learning Representations*, 2024.

[14] Kushal Arora, Layla El Asri, Hareesh Bahuleyan, and J. Cheung. Why Exposure Bias Matters: An Imitation Learning Perspective of Error Accumulation in Language Generation. *Findings*, April 2022.

[15] Xue Jiang, Yihong Dong, Yongding Tao, Huanyu Liu, Zhi Jin, and Ge Li. Rocode: Integrating backtracking mechanism and program analysis in large language models for code generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 334–346. IEEE, 2025.

[16] Boaz Lavon, Shahar Katz, and Lior Wolf. Execution Guided Line-by-Line Code Generation, June 2025.

[17] David Brandfonbrener, Simon Henniger, Sibi Raja, Tarun Prasad, Chloe Loughridge, Federico Cassano, Sabrina Ruixin Hu, Jianang Yang, William E. Byrd, Robert Zinkov, and Nada Amin. VerMCTS: Synthesizing Multi-Step Programs using a Verifier, a Large Language Model, and Tree Search, May 2024.

[18] Pranjal Aggarwal, Bryan Parno, and Sean Welleck. AlphaVerus: Bootstrapping Formally Verified Code Generation through Self-Improving Translation and Treefinement, December 2024.

[19] Qwen Team. Qwen3 Technical Report, May 2025.

[20] Clement Farabet and Olivier Lacombe. Gemma 4: Byte for byte, the most capable open models, April 2026.

[21] Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E. Gonzalez, and Ion Stoica. S*: Test Time Scaling for Code Generation. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 15964–15978. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-emnlp.865.

[22] Ruchi Puri, David S. Kung, G. Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir A. Zolotov, Julian Dolby, Jie Chen, M. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, and Ulrich Finkler. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. *NeurIPS Datasets and Benchmarks*, May 2021.

[23] Ming-Ho Yee and Arjun Guha. Do Machine Learning Models Produce TypeScript Types That Type Check? *LIPIcs, Volume 263, ECOOP 2023*, 263:37:1–37:28, 2023. ISSN 1868-8969. doi: 10.4230/LIPIcs.ECOOP.2023.37.

[24] Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, L. Wang, Antonia Creswell, G. Irving, and I. Higgins. Solving math word problems with process- and outcome-based feedback, November 2022.

[25] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training Verifiers to Solve Math Word Problems, October 2021.

[26] Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. Math-Shepherd: Verify and Reinforce LLMs Step-by-step without Human Annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9426–9439. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.510.

[27] Liangchen Luo, Yinxiao Liu, Rosanne Liu, Samrat Phatale, Harsh Lara, Yunxuan Li, Lei Shu, Yun Zhu, Lei Meng, Jiao Sun, and Abhinav Rastogi. Improve Mathematical Reasoning in Language Models by Automated Process Supervision, June 2024.

[28] Qinglin Wang, Zhihong Sun, Ruyun Wang, Tao Huang, Zhi Jin, Ge Li, and Chen Lyu. Semguard: Real-time semantic evaluator for correcting llm-generated code. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1919–1930. IEEE, 2025.

[29] Poorva Garg, Renato Lui Geh, Daniel Israel, Todd Millstein, Kyle Richardson, and Guy Van den Broeck. Probabilistic Programs of Thought, April 2026.

[30] Alexander Du, Jianjun Ou, Danyang Zhuo, and Matthew Lentz. Hydra: Efficient, correct code generation via checkpoint-and-rollback support. *arXiv preprint arXiv:2605.15238*, 2026.

[31] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.emnlp-main.779.

[32] Gabriel Poesia, Oleksandr Polozov, Vu Le, A. Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. In *International Conference on Learning Representations*, January 2022.

[33] Brandon T. Willard and Rémi Louf. Efficient Guided Generation for Large Language Models, July 2023.

[34] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting Is Programming: A Query Language for Large Language Models. *Proceedings of the ACM on Programming Languages*, 7 (PLDI):1946–1969, June 2023. ISSN 2475-1421. doi: 10.1145/3591300.

[35] Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 10932–10952. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.674.

[36] Lakshya Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K Lahiri, and Sriram Rajamani. Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context. In *Advances in Neural Information Processing Systems*, 2023.

[37] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, pages 172–184, New York, NY, USA, November 2023. Association for Computing Machinery. ISBN 979-8-4007-0327-0. doi: 10.1145/3611643.3616271.

[38] Niels Mündler, Jingxuan He, Hao Wang, Koushik Sen, Dawn Song, and Martin Vechev. Type-Constrained Code Generation with Language Models. In *Programming Language Design and Implementation*, May 2025. doi: 10.1145/3729274.

[39] Debangshu Banerjee, Tarun Suresh, Shubham Ugare, Sasa Misailovic, and Gagandeep Singh. CRANE: Reasoning with constrained LLM generation. In *International Conference on Machine Learning*, February 2025.

[40] Henrijs Princis, Arindam Sharma, and Cristina David. TreeCoder: Systematic Exploration and Optimisation of Decoding and Constraints for LLM Code Generation, November 2025.

[41] Timothy J. Boerner, Stephen Deems, Thomas R. Furlani, Shelley L. Knuth, and John Towns. ACCESS: Advancing innovation: NSF's advanced cyberinfrastructure coordination ecosystem: Services & support. In *Practice and Experience in Advanced Research Computing 2023: Computing for the Common Good*, PEARC '23, pages 173–176, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399852. doi: 10.1145/3569951.3597559. URL https://doi.org/10.1145/3569951.3597559.

[42] Anh Ta, Junjie Zhu, and Shahin Shayandeh. Reinforced Agent: Inference-Time Feedback for Tool-Calling Agents, April 2026.

[43] Tianyang Zhou, Ziyi Zhang, Haowen Lin, Somesh Jha, Mihai Christodorescu, Kirill Levchenko, and Varun Chandrasekaran. SACTOR: LLM-Driven Correct and Idiomatic C to Rust Translation with Static Analysis and FFI-Based Verification, March 2025.

[44] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In *Lecture Notes in Computer Science*, pages 257–281. Springer Berlin Heidelberg, 2014. ISBN 978-3-662-44201-2 978-3-662-44202-9. doi: 10.1007/978-3-662-44202-9_11.

[45] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 152–162. ACM, October 2018. doi: 10.1145/3236024.3236051.

APPENDIX

A Limitations

DTV rests on assumptions that bound its applicability and incurs non-trivial per-task and per-model implementation effort, both of which we document here.

Oracle requirements. DTV requires oracles to produce *local, actionable* diagnostics with source-line attribution (Section 2.2); oracles returning only a global pass/fail verdict can drive the commit decision but cannot guide structure-aware rollback or feedback. Our in-loop oracle suite consists only of compilation-style verifiers (`rustc` on C to Rust, `tsc` and ESLint on JS to TS); slower or weaker-signal oracles such as cross-language differential testing (Table 3) are reserved for post-hoc evaluation, and whether DTV's in-loop mechanism remains effective when driven by such oracles is not addressed empirically.

Non-local semantic errors. Structure-aware rollback is most effective when a failure can be fixed within a bounded region. An upstream edit may still be reachable through block/function escalation, as the analyses in Section 4.2 illustrate, but cross-module type inconsistencies or architectural redesign can require broader changes. The JS-to-TS inner oracle filters TS2322, TS2339, TS2551, and TS2345 when they are not actionable under partial context; all remain in the shared final gate. Gains persist under additional strict type checking (Appendix J.4), while the absolute pass rates show that annotation coverage alone does not establish full type correctness.

Implementation and evaluation scope. DTV requires language-specific rendering and context rules, verifier drivers and diagnostic filters, and model-family decoding adapters. Evaluation covers Qwen3-4B, Gemma-4-E4B, and the 30.5B-total-parameter Qwen3-30B-A3B MoE on two translation tasks. Gains with the stronger model expand the tested range, but do not establish generality across model scales, dense architectures, or non-translation domains. Near-saturation baseline performance can also limit the benefit of in-loop recovery, as seen on Gemma C to Rust.

Fixed recovery policy. The current policy uses fixed retry and bailout thresholds (Appendix H). The ablations support recovery and scope widening, but do not show that these thresholds are optimal. Conditioning escalation and bailout on remaining budget and diagnostic type is a natural extension. The observed-edit analyses use successful-program diffs as localization proxies rather than identifying necessary or unique repairs.

Verifier wall-clock cost. Token matching controls generated-token expenditure rather than total runtime. Our measurements include verifier cost: C-to-Rust runs save wall-clock time, while the serial TypeScript/ESLint stack adds overhead (Table 6). Relative overhead decreases in the stronger-model experiment, consistent with a more favorable balance when generation is expensive relative to verification. Toolchain startup, code size, and CPU resources can change this balance, so token-Pareto improvements need not imply wall-clock improvements for every verifier stack.

B Broader Impacts

DTV is a decoding-time algorithm for code translation under deterministic verifiers; it does not train new models or release new generative capabilities. Its most direct positive impact is lowering the cost of translating legacy code into safer target languages, for example C to Rust, where memory-safety bugs account for a substantial fraction of reported security vulnerabilities, and JavaScript to TypeScript, where static typing catches a class of latent defects at development time. By converting outcome verifiers into process-level guidance, DTV also reduces the number of generation tokens spent on rejected candidates relative to outcome-only decoding, which lowers compute and energy cost per successful translation.

The most direct negative impact is the risk of misplaced confidence in oracle-passing translations. DTV's COMMIT decision certifies only that the committed prefix passes the configured verifier suite (compilation, type checking, and differential testing on a finite set of inputs). It does not certify behavioral equivalence on all inputs, freedom from undefined behavior, concurrency correctness,

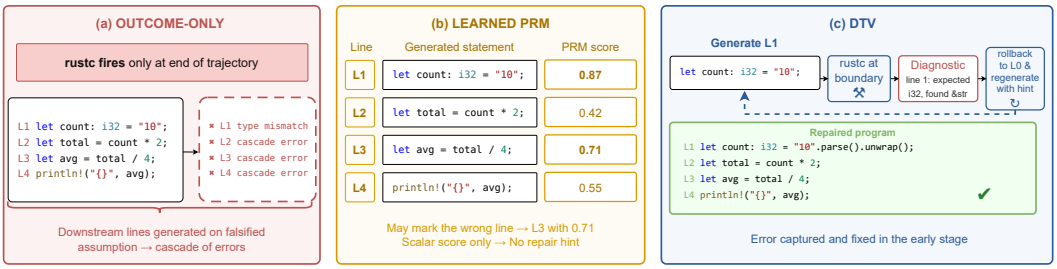


Figure 4: Three verification paradigms on an illustrative C-to-Rust translation. **(a)** Outcome-only verifies after completion; **(b)** a learned PRM produces scalar ranking signals; **(c)** DTV verifies at structural boundaries, rolls back, and regenerates with diagnostic feedback.

or fitness for any particular safety-critical use. Users who deploy DTV-translated code without independent review, especially in security-sensitive or safety-critical settings, may be exposed to defects outside the oracles' coverage. We recommend treating DTV outputs as a starting point for human review and co-designing the verifier suite with the deployment risk profile (e.g., adding fuzzing, sanitizers, or formal verification for high-assurance settings).

C Verification Paradigms

Figure 4 situates DTV against the two paradigms that dominate prior work on verifier-guided generation. Outcome-only verification (panel a) applies verifiers post-hoc to completed samples, so a failing verdict discards an entire program-length sequence and the model has already extended downstream code on top of the falsified prefix. Learned process supervision (panel b) attaches probabilistic scores to intermediate steps, but a high score cannot certify that a partial translation can be extended to pass the compiler, type checker, and differential tests, and training such scorers requires verifier-, language-, and source-target-specific supervision. DTV (panel c) integrates deterministic verifiers directly into the autoregressive loop, combining structural-boundary verification, structure-aware rollback, and feedback re-injection. Appendix D discusses the closest prior work in each paradigm.

D Related Work (Extended)

This appendix expands the compressed discussion in §5 with a per-system breakdown.

Test-time compute scaling. Test-time compute scaling has emerged as a complement to parameter scaling: extra inference compute can substitute for larger models. Existing approaches differ along two axes: the *signal source* used to allocate compute (repeated independent sampling [7], compute-optimal allocation against learned process reward models [1], deliberate search over reasoning traces [8], and end-to-end long-reasoning systems [2, 5]; surveys catalog these strategies [3]), and the *decoding integration* of those signals (outcome-only after a complete program, or process-level inside generation). DTV is itself a test-time compute scaling method, in the *process-level supervision* sub-family, with the distinguishing feature that signals come from deterministic verifiers (compilers, type checkers, tests) integrated at structural boundaries during decoding. Probabilistic programs of thought (PPoT) [29] are also motivated by fixed-budget inference: they reuse next-token probabilities from completed LLM samples to draw cheap local variants before post-hoc verification, whereas DTV changes when verifier diagnostics affect the search by committing, rolling back, and repairing prefixes. The rest of this section compares DTV against the closest prior work in each adjacent corner: learned process scorers, execution-guided and verifier-guided generation (post-hoc and in-loop), and constrained decoding.

Process supervision via learned scorers. Process reward models (PRMs) train a learned scorer on step-level annotations and outperform outcome-only verifiers on multi-step reasoning [4, 24, 25]. Recent work reduces annotation cost via automated step supervision [26, 27], while Khalifa et al. [10] replace discriminative PRMs with chain-of-thought verifiers. Reinforced Agent [42] similarly moves

feedback before tool execution, its signal is a reviewer LLM over complete tool-call actions rather than a deterministic verifier over rendered code artifacts. DTV produces step-level signals as well, but obtains them from *deterministic oracles* rather than learned scorers. This shifts the design space along three axes: signals are *exact* rather than *probabilistic*, enabling hard COMMIT and ROLLBACK actions instead of soft re-ranking; they require *no annotation or training data*; and they come with *structured diagnostics* (error messages, failing locations) that DTV uses for repair—none of which is naturally available from a learned PRM.

Execution-guided and verifier-guided generation. One line of work uses deterministic verifiers (compilers, tests, formal solvers) to evaluate complete programs after generation, using the verdict for filtering, ranking, or single-shot repair [6, 11–13, 21]. A more recent line moves the verifier *inside* generation: Lavon et al. [16] run line-by-line Python execution feedback through Classifier-Free Guidance (EG-CFG); Brandfonbrener et al. [17] use MCTS guided by partial-program verification in Dafny and Coq (VerMCTS), with a token-budget metric (pass@T) closely related to ours; and Aggarwal et al. [18] add verifier-driven tree-search refinement (Treefinement) for translating Dafny to Verus (AlphaVerus). DTV differs from these in-loop methods in five respects: (i) *adaptive structural-boundary granularity* spanning statement, block, function, and program, rather than a single fixed scope (line in EG-CFG, function in VerMCTS), motivated by the fact that partial programs in static-typed mainstream languages admit only specific oracle-consumable scopes; (ii) *hard state-machine control* with explicit COMMIT, ROLLBACK, and FEEDBACK actions, rather than soft probability interpolation, stochastic search, or whole-program restart; (iii) *multi-oracle composition* (compiler diagnostics, type checker, differential testing) rather than a single signal type; (iv) *decoding-time only* with a frozen pretrained model, while AlphaVerus involves a self-improving training loop; and (v) *structured diagnostic-driven repair* via the explicit FEEDBACK action and its patch-based retry mode, which consume oracle diagnostics and failing locations to construct targeted repair prompts, rather than passing raw execution traces as soft prompt context (EG-CFG) or discarding the failure content beyond a scalar pass/fail signal (VerMCTS, AlphaVerus).

ROCODE and source-conditioned repair. ROCODE [15] establishes program-analysis-triggered backtracking during code synthesis. For a new located error it rolls back to the reported line; repeated or unlocated errors invoke a global maximum-token-entropy statement rule, while persistent syntax errors can trigger full restart. Its constrained regeneration stores failed paths in a trie and applies position-dependent probability decay along those paths. Analyzer diagnostics determine whether and where to backtrack, but their text is not added to the model's regeneration context.

DTV shares the in-loop premise and addresses context preservation through explicit structural repair regions. A source program and retained target prefix continue to condition retries, so discouraging one token path does not itself specify the target-language constraint to repair. DTV supplies diagnostic text and widens the editable region when local retry stalls. The ownership example in § 2.2 illustrates why a later diagnostic may require reopening an earlier decision; Appendix N.1 compares diagnostic conditioning with ROCODE path decay, and Appendix N.2 evaluates its entropy fallback on observed upstream edits.

The released ROCODE adapter makes newline-terminated Python prefixes analyzable by inserting `pass`/`break` and ignoring selected EOF failures. Its paper also reports C++ results, but the corresponding partial-artifact construction and Rust/TypeScript integrations are not provided in the released artifact. A cross-language port therefore requires additional rendering and compiler-driver choices. The mechanism analyses in Appendix N use a common Rust renderer and oracle to isolate regeneration and localization choices from language-adapter differences.

Concurrent work, Hydra [30], targets the cost of repairing static errors in C/C++ code generation. It streams generated code into a modified Clang front end that checks prefixes asynchronously and checkpoints checker state by forking the compiler process, and it selects rollback points with a root-cause prior fitted on a training split of its benchmark. Relative to post-hoc repair, Hydra reduces latency and token consumption, while both reach near-complete static correctness. DTV differs in two respects. First, it requires no verifier modification. Hydra's retrofit relies on a checker that performs semantic analysis during parsing, as Clang does. DTV instead runs unmodified verifiers on rendered artifacts, including non-compiler oracles such as ESLint. This matters for `rustc`, which parses the whole crate before type checking function bodies and borrow checking their MIR: a streaming front end would not reach these checks until the input is complete, whereas rendering exposes them at

statement boundaries. Second, it targets pass rate under a fixed per-case token budget rather than the cost of reaching an outcome that post-hoc repair also attains. Its baselines remain far from saturated (with Qwen3-4B, naïve/self-refine reaches 72.3% on C to Rust and 33.3% on JS to TS), so in-loop verification yields higher pass rate at matched cost rather than only lower cost.

Constrained decoding for code. Constrained decoding enforces *hard* token-level constraints derived from a recognizer (grammar, regex, finite-state machine, or type automaton) that operates over partial token sequences, masking tokens that would violate the recognizer at each sampling step [31–35]. Recent code-aware variants extend this in several directions: Agrawal et al. [36] drive decoding with repository-level static analysis; Wei et al. [37] prune tokens via a completion engine; Mündler et al. [38] introduce prefix automata over inhabitable types; and Banerjee et al. [39] augment the grammar to preserve reasoning capacity. More recent framework work generalizes this with pluggable search strategies and composable constraints: Princis et al. [40] unify sampling, beam search, MCTS, SMC, and ASAp under a tree-search abstraction that admits both token-level syntactic constraints and whole-program execution constraints (e.g., unit tests) via a product-of-experts formulation. By construction, these methods enforce constraints either as recognizers over partial token sequences (typically syntax or local typing) or as post-hoc filters on completed programs; they cannot invoke semantic verifiers at intermediate program scopes during generation. DTV is orthogonal in two respects. First, it accepts oracles as black boxes, including those that require structurally complete artifacts and therefore cannot be expressed as token-level recognizers (compilation against the full type and borrow checker, differential testing against the source program). Second, it does not constrain the base sampler; verification failures are handled *after the fact* by structure-aware rollback and diagnostic feedback, rather than prevented up front. The two approaches compose: constrained decoding can enforce hard syntactic invariants while DTV adds a semantic verification layer on top, a combination we leave to future work.

In summary, DTV sits within test-time compute scaling and process-level supervision and shares the in-loop verification premise of ROCODE, EG-CFG, and VerMCTS. Its architectural commitments are deterministic verifier signals (vs. learned PRMs), adaptive structural-boundary granularity with explicit checkpoint rollback (vs. fixed-scope or whole-program in-loop methods), nested scope escalation with diagnostic-conditioned regeneration (vs. ROCODE's line- or entropy-selected rollback with failed-path decay), and a hard state-machine controller composing multiple oracles (vs. soft re-ranking, syntactic recognizers, or training-time verifier-feedback pipelines).

E Task and Oracle Setup

This appendix expands the per-task descriptions, dataset construction pipelines, and oracle configurations summarized in Section 3.

C to Rust. A representative unsafe-to-safe systems translation task. C programs routinely rely on manual memory management and undefined behavior that must be resolved into explicit Rust idioms (ownership, borrowing, lifetimes), making correctness verification non-trivial and the task widely studied in the program-analysis community [43]. The Rust compiler (`rustc`) provides a strong static oracle spanning syntactic and semantic analysis (types, borrows, lifetimes), and differential testing, executing both the C source and the Rust translation on the same inputs and comparing outputs, provides a behavioral equivalence oracle. These oracles complement each other: `rustc` supplies static signals on partial programs, while differential testing supplies behavioral signals on complete programs.

JavaScript to TypeScript. A representative dynamic-to-static typing task. The translation is primarily about introducing sound type annotations: when an explicit annotation is missing and the compiler's inference cannot resolve a type, TypeScript silently falls back to *implicit any*, which passes compilation but bypasses the type system and exposes the program to the type-related safety issues that migration was meant to eliminate [44, 45]. Catching such degradation requires verifiers beyond the compiler alone. Two verifiers are used: the TypeScript compiler (`tsc`) for compilation checks, and ESLint for detecting missing type annotations. Unlike C to Rust, TypeWeaver packages do not ship with executable test suites, so differential testing is not available; the compiler and linter serve as the sole correctness oracles.

Component instantiation. Table 3 summarizes how the abstract DTV components (Section 2.2) are instantiated for each task. The two tasks span complementary verification axes: unsafe-to-safe (C to Rust) versus dynamic-to-static (JS to TS), and compiler-plus-executable-test versus compiler-plus-linter oracle suites.

Table 3: Instantiation of DTV components for each translation task. The in-loop oracle drives DTV's in-generation verification, rollback, and feedback. The outer-loop oracle judges pass/fail at the end of each generation attempt for self-refine and best-of-N. The post-hoc oracle is used only for final evaluation.

Component	C to Rust	JS to TS
Renderer	Rust structural completion	TS delimiter + context rules
In-loop oracle	`rustc`	`tsc` (weakened) + ESLint
Outer-loop oracle	`rustc`	`tsc` (strict off) + ESLint
Post-hoc oracle	`rustc`; differential testing	`tsc` (strict off) + ESLint

C to Rust dataset and test-input generation. We construct the 300-sample evaluation set in three stages, all using fixed random seeds. *Stage 1: pool construction.* We filter Project CodeNet [22] to C submissions with status `Accepted` and retain only those whose source contains a standard-input call (`scanf`, `fgets`, `getchar`, etc.), since differential testing requires programs that consume external inputs. For each problem we select the submission with median code size as its representative, ensuring at most one program per problem. *Stage 2: stratified sampling.* We stratify the per-problem representatives by source byte size into three tiers (small $<$300 bytes, medium 300–1000 bytes, large $>$1000 bytes) and sample proportionally across tiers with a minimum of 10 representatives per tier, yielding 400 candidate programs. *Stage 3: test-input generation.* CodeNet does not provide executable test inputs, so we generate them per candidate via a hybrid LLM-seeded AFL++ pipeline. We first prompt GPT-5.4 for 25 stdin test inputs per program, with instructions to include boundary values and ASCII-only content, and validate each candidate by executing the AFL-instrumented binary; seeds that exit non-zero or exceed a 5-second timeout are discarded. The validated seeds bootstrap AFL++ fuzzing for 90 seconds in stdin mode. The resulting AFL++ queue is minimized with `afl-cmin` and further filtered through AddressSanitizer, UndefinedBehaviorSanitizer, and Valgrind `memcheck` to drop inputs that trigger crashes, undefined behavior, or memory errors; surviving inputs form the final test corpus per program. The pipeline succeeds on 306 of the 400 candidates; the remaining 94 fail due to compilation errors, lack of valid LLM seeds after execution filtering, AFL++ producing no queue entries, or no inputs surviving the sanitizer filter. We draw 300 cases uniformly at random from these 306 as the final evaluation set. Coverage is measured by re-executing each surviving input against a coverage-instrumented `clang` build and reported via `llvm-cov` source-based coverage. The resulting fuzz corpora attain mean per-sample line coverage of 93.4% (aggregate 85.1%, 11,721 of 13,781 lines) and mean per-sample branch coverage of 87.5% (aggregate 78.9%, 6,838 of 8,662 branches). The selected programs span 16–448 lines of code, with median 46 and mean 68.8.

JavaScript to TypeScript dataset. We construct the 150-sample evaluation set from TypeWeaver [23]. We start from the two subsets `top1k-typed-nodeps-es6` and `top1k-untyped-nodeps-es6` and process each candidate package as follows. For each package we produce a single bundled JavaScript file via `rollup` (ES module format on the package's main entry). We discard packages whose bundled size falls outside $[30, 1000]$ lines of code, and we exclude packages that already pass `tsc` under strict mode without modification, since these provide no translation signal. From the resulting pool we sample 150 packages uniformly at random (seed 42). The selected packages span 30–741 bundled lines of code, with median 91 and mean 149.2.

Oracle diagnostic filtering. All three in-generation verifiers (`rustc`, `tsc`, and ESLint) filter their raw diagnostic output before it drives rollback or feedback in DTV. We distinguish two categories of filtering.

Partial-context filters. Partial-code checks can report incompleteness, such as missing definitions or closing syntax, before generation supplies the remaining context. The intermediate oracle drivers therefore apply the following filters:

- **rustc (C to Rust):** suppresses unresolved type, value/function, and module diagnostics (E0412, E0425, E0433) in the partial-program oracle. It also suppresses E0277 trait-bound diagnostics with an attached replacement suggestion or an `impl` header, allowing later generated declarations to supply missing context.
- **tsc (JS to TS):** suppresses selected missing-name, missing-module, and initialization diagnostics (TS2304, TS2552, TS2307, TS2564) while checking partial translations.
- **ESLint (JS to TS):** drops diagnostics whose source position lies past the current prefix boundary, which can arise because our rendered artifact includes context-rule additions (e.g., a closing }) beyond the actual generated prefix.

TypeScript settings and inner type-error filtering. The JS-to-TS instantiation uses the following compiler settings:

- **Shared strictness setting:** both inner checks and the common final gate invoke tsc with `strict: false`, suppressing strictness-family diagnostics (including `noImplicitAny`).
- **Type-correctness error filter:** TS2322 (assignment type mismatch), TS2339 and TS2551 (missing property, with or without a suggested alternative), and TS2345 (argument type mismatch) are dropped before the inner verdict is computed.

These errors are common in partial translations but are typically not actionable for in-loop retry: they often reflect missing context about external types or deliberate typing choices that become resolvable only once the full program is available. Retaining them would cause DTV to spend retry budget on diagnostics the model cannot productively fix. We therefore filter them during generation only; the outer-loop and post-hoc tsc oracles preserve them and let them count against the verdict.

The shared final gate invokes the compiler and linter directly, retaining all of these error classes for every method. It uses `strict: false` and the ESLint `@typescript-eslint/typedef` rule to require explicit annotations. Annotation presence and strict type checking are complementary criteria; the latter also checks additional consistency and strictness properties. Appendix J.4 therefore reports an additional shared strict recheck of saved outputs.

The filters determine when the controller receives actionable process feedback; they do not relax the common final acceptance criterion. Final C-to-Rust acceptance similarly uses unfiltered `rustc`, with differential testing reserved for behavioral evaluation.

Baseline configuration details. S* [21] is configured with $N{=}8$ parallel candidates, $R{=}3$ compile-feedback rounds per candidate, and compile-error argmin selection (the candidate whose final round compiles with the fewest errors is returned, breaking ties by sample order). Both choices deviate from the original defaults ($N{=}16$, $R{=}2$, adaptive input synthesis) for two reasons. First, $N{=}8, R{=}3$ replaces $N{=}16, R{=}2$ because $N{=}16$ would exceed the per-case token cap under which DTV and naïve/BoN-32 operate; the resulting per-case generation budget of $N \times R = 24$ (versus the paper's default of 32) aligns with the envelope shared by the other baselines. Second, compile-error argmin selection replaces adaptive input synthesis because the original mechanism prompts an auxiliary LLM to synthesize distinguishing test inputs and selects the candidate whose execution outputs align with the LLM's predicted behavior, a procedure that presupposes the availability of program-level test inputs in the task specification, which translation tasks (C-to-Rust, JS-to-TS) do not provide. We therefore adopt the "Public Only" selection variant reported in Table 3 of Li et al. [21], which selects the candidate whose final round compiles with the fewest errors and breaks ties by sample order. All methods share the final compiler/linter acceptance criterion; the intermediate diagnostic filtering described above is specific to DTV's search. Because BoN-32 and S* are substantially more computation-intensive than the head-to-head configurations, the BoN grid, S* comparisons, and Gemma cross-model analysis (Appendix K) use $n = 200$ for C to Rust and $n = 100$ for JS to TS. Qwen3-30B-A3B uses the first 100 cases per task; the ablations and mechanism comparisons use the first 100 C-to-Rust cases. These subsets all come from the main evaluation pools.

F Model and Decoding Hyperparameters

All experiments are run locally via the HuggingFace Transformers library. We evaluate the following open-weight instruction-tuned models:

- **Qwen3-4B-Instruct** (`Qwen/Qwen3-4B-Instruct-2507`) [19].

- **Gemma-4-E4B-it** (`google/gemma-4-E4B-it`) [20].
- **Qwen3-30B-A3B-Instruct** (`Qwen/Qwen3-30B-A3B-Instruct-2507`), a 30.5B-total-parameter mixture-of-experts model, at revision `0d7cf23991f47feeb3a57ecb4c9cee8ea4a17bfe` [19].

All models are used as black-box generators (no fine-tuning). DTV and the naïve baseline share the same model and decoding configuration within a given experiment, so any pass-rate difference isolates the effect of the decoding strategy rather than the underlying sampler.

Sampling. Between meta steps we use each model's default sampler as shipped in its `generation_config.json`, without overriding sampling hyperparameters. Table 4 lists the values for all evaluated models. A greedy decoding mode (`do_sample=False`) is exposed as an ablation option but is not used in the main experiments.

Table 4: Default sampler hyperparameters shipped with each model's `generation_config.json`.

Model	`do_sample`	`temperature`	`top_k`	`top_p`
Qwen3-4B-Instruct-2507	true	0.7	20	0.8
Gemma-4-E4B-it	true	1.0	64	0.95
Qwen3-30B-A3B-Instruct-2507	true	0.7	20	0.8

Generation caps. The boundary-level generator requests at most 1024 new tokens per invocation, stopping earlier at a structural boundary or the remaining token budget. Inner controller actions and outer refinement rounds each have a safety cap of 2000; these count different operations. Naïve complete-program generation is bounded by the remaining case budget. The per-case generation-token budget is set to $16\times$ the source token count (Section 3), consumed across all generation within a case (including tokens discarded by rollback).

Hardware. The Qwen3-4B and Gemma evaluation uses a single Nvidia A100 (40GB) or GH200 (96GB) GPU per run, with BF16 model weights. Qwen3-30B-A3B uses a 96GB H100 GPU with eight Grace ARM CPU cores. Compiler and linter subprocesses run on the CPU; generation runs on the GPU. The matched regeneration and entropy replays use an RTX 3090 with BF16 Qwen3-4B, PyTorch 2.10.0+cu126, and Transformers 5.5.4. Appendix J.3 defines the reported timing interval.

G Per-Action Semantics

The controller's action set is implemented as follows; the scope, retry-mode, and bailout policies are described in Appendix H.

- GENERATE: invoke the base sampler to extend the current prefix by one or more tokens, stopping at a structural boundary (e.g., end of statement) or when the generation budget is exhausted.
- VERIFY: render the current prefix into an artifact, select applicable oracles by scope, and record their verdicts and diagnostics into the feedback state.
- COMMIT: mark the current prefix as a trusted checkpoint, saving the group stack state from the renderer for later structure-aware rollback.
- ROLLBACK: restore a checkpoint at scope $\ell \in \mathcal{G}$; statement rollback restores the last committed checkpoint, block/function rollback truncates to the group start and drops affected frames, and program rollback clears all checkpoints.
- FEEDBACK: construct a retry prompt from the current feedback state and ask the model to address the diagnostics; supports two realizations described in Section 2.2.
- TERMINATE: end the decoding loop; triggered normally at end-of-sequence, or early in the case of a *bailout* when the same diagnostic persists across scope escalations.

H Implementation Details

This appendix specifies the concrete data structures and policies that instantiate the DTV algorithm of Section 2.2. Other instantiations are possible; we describe the one used in our experiments.

Scope map implementation. We implement the scope map $g : \mathcal{A} \to \mathcal{G}$ via a *group stack* maintained by the renderer: as tokens stream in, the renderer pushes frames onto the stack at block and function

Algorithm 1 Decoding Time Verification (DTV). Given source S, oracles $\mathcal{O}$, and budget B_{gen}, DTV alternates boundary-aligned token generation with oracle verification. Failures trigger feedback-augmented rollback; success terminates at program-level oracle pass.

Require: source S, oracles $\mathcal{O}$, generation budget B_{gen}
Ensure: terminal artifact A, success flag $s \in \{\text{SUCC}, \text{FAIL}\}$
$Y \leftarrow \emptyset,\ \mathcal{C} \leftarrow \{\emptyset\},\ \mathcal{F} \leftarrow \emptyset$ {init prefix, checkpoints, feedback state}
while $N_{\text{gen}} < B_{\text{gen}}$ **do**
  $Y \leftarrow \textsc{Generate}(Y,\ \mathsf{Augment}(S, \mathcal{F}))$ {sample to next structural boundary or EOS}
  $A \leftarrow \mathsf{Render}(S, Y);\ \{(q_j, d_j)\} \leftarrow \textsc{Verify}(A, \mathcal{O})$
  if $q_j = 1$ for all applicable j **then**
    $\mathcal{C} \leftarrow \textsc{Commit}(\mathcal{C}, Y)$
    if $g(A) = \texttt{program}$ **then**
      return $(A,\ \text{SUCC})$
    end if
  else
    $\mathcal{F} \leftarrow \textsc{Update}(\mathcal{F},\ \{(q_j, d_j)\})$
    $(\mathit{action},\ \ell) \leftarrow \textsc{Decide}(\mathcal{F})$
    if $\mathit{action} = \text{BAILOUT}$ **then**
      return $(A,\ \text{FAIL})$
    end if
    $Y \leftarrow \textsc{Rollback}(\mathcal{C}, \ell)$
  end if
end while
return $(\mathsf{Render}(S, Y),\ \text{FAIL})$ {budget exhausted}

openings, and pops them at the corresponding closings. This mirrors the scope-tracking stack standard parsers maintain when reading source code, with the difference that it operates on the partial generation rather than on a complete file. The renderer is paired with a token-level stopping criterion that halts the base sampler when the current prefix reaches a structural boundary (with awareness of strings, comments, and nested delimiters), handing control back to the controller to trigger the next meta step.

The renderer supplies the oracle-consumable artifact referenced by the legal prefix invariant (Section 2.2): it emits only when the current prefix ends at a structural boundary and the group stack is in a consistent state. The semantic side of the invariant (absence of oracle violations at the current scope) is decided by VERIFY and COMMIT in Algorithm 1.

Checkpoints and repair scheduling. Each COMMIT checkpoints the current prefix and synchronizes the renderer's group stack. Statement rollback restores the last committed checkpoint without deleting it. Block/function rollback truncates to the corresponding group start and drops the invalidated frames. The manager also supports program rollback, which clears the checkpoints, but the evaluated inner policy caps rollback at function scope and delegates whole-program retries to the outer loop. Retry has a *scope* $\ell \in \{\texttt{stmt}, \texttt{block}, \texttt{func}\}$ and a *mode* (Appendix I):

- *Inline continuation*: the model resumes generation from the rolled-back prefix with the diagnostic summary appended as an inline comment.
- *Patch-based repair*: instead of continuing inline, the controller closes the current assistant turn and asks the model in a new user turn for a minimal patch addressing the diagnostics. The patch is constrained to the rollback scope; outputs that violate the constraint are rejected and handled by the recovery policy.

The scope and mode schedules use distinct counters. Scope escalation is a fixed heuristic over repeated local diagnostic stalls: rollback starts at statement scope, and when the same diagnostic persists beyond a fixed retry count, rollback widens to the nearest active block, or function if no block is active. Oracles may also request a coarser scope directly, subject to the function-scope cap; the in-loop oracles in our experiments do not, so rollback in all reported runs starts at statement scope. A successful or inconclusive verification resets the statement-stall counter.

Feedback scheduling is keyed by the restored prefix and repair scope. A new key starts with inline continuation. Persistent failure switches to patch repair; function-scope failures or a high diagnostic-

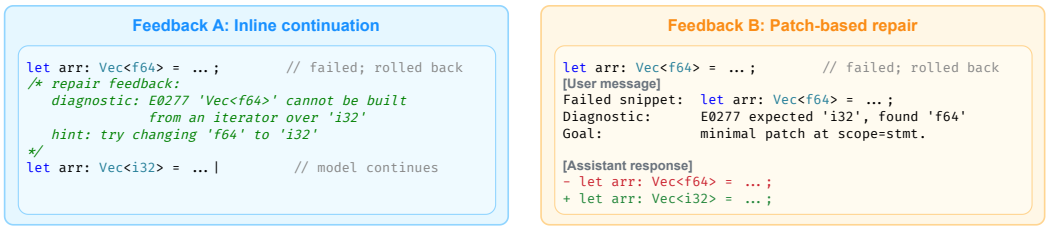


Figure 5: The two repair modes (Section 2.2) on the same `rustc` E0277 failure. **(left)** *Inline continuation*: diagnostic appended as a comment, model continues in the same turn. **(right)** *Patch-based repair*: structured user turn, model returns a diff patch in a fresh assistant turn. Diagnostics and hints are shown in simplified form.

error count can trigger this switch earlier, after an inline attempt. Once patch repair is selected, it stays selected for that key. Repeated responses without a usable patch invoke rollback and its scope-selection policy. A changed prefix/scope creates a new repair key, so mode can return to inline after widening.

Bailout. The controller counts verified failures at each diagnostic anchor (enclosing function, primary error code, primary message), independent of the current rollback scope. At the configured visit limit, it issues TERMINATE with bailout flag $\beta_k = 1$ before the next rollback, returning the current artifact and diagnostics. Counters reset for a new outer attempt. Budget exhaustion and candidate closure also end the inner run, and the common final gate evaluates the returned artifact. Thus bailout can occur before every repair mode has been exhausted. Recovery settings were tuned on a small C-to-Rust sample and then held fixed for all reported experiments, including JS to TS and the other generators; the ablations establish the value of recovery and escalation, not the optimality of these settings.

Reusable core and adapters. The controller, rollback, feedback, shared types, and oracle interfaces form the reusable core. Language integration implements prefix rendering, context rules, and feedback configuration; verifier integration implements drivers and diagnostic handling; model-family backends implement decoding and chat-format control.

I Feedback Prompt Formats

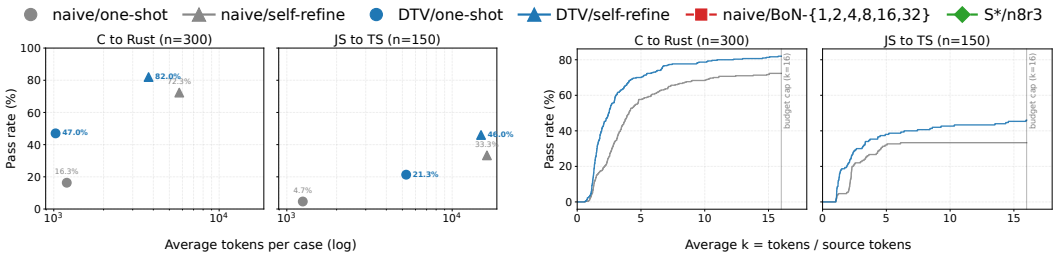
The two retry modes from Section 2.2 differ in how the diagnostic-augmented prompt is delivered to the model and what the model is asked to return. Figure 5 shows the same C-to-Rust failure (a `rustc` E0277 trait-bound error: the model bound `Vec<f64>` where the surrounding context required `Vec<i32>`) handled by each mode.

J RQ1 detailed results

This appendix complements RQ1 (Section 4.1) with full results, paired tests, the BoN comparison, measured runtime, and strict rechecking.

J.1 Full-set head-to-head comparison

Figure 6 reproduces main-body Figure 2 on the full evaluation set ($n = 300$ for C to Rust, $n = 150$ for JS to TS), restricted to the four head-to-head configurations referenced in Section 4.1 (naïve vs. DTV with one-shot vs. self-refine). The cost-matched subset version, jointly comparing these head-to-head configurations against the BoN-N and S* baselines on the $n = 200$ C-to-Rust and $n = 100$ JS-to-TS subsets, is reported in main-body Figure 2; the head-to-head pass rates on the full set quoted in Section 4.1 are taken directly from the data underlying this appendix figure.



(a) Per-case avg tokens (log scale) vs. compile pass rate.

(b) Cumulative pass rate vs. per-case cost k.

Figure 6: Full-set head-to-head comparison on C to Rust ($n = 300$) and JS to TS ($n = 150$), restricted to the four head-to-head configurations (naïve vs. DTV with one-shot vs. self-refine); the legend is shared with main-body Figure 2 (BoN-N and S* entries do not appear in this figure). **(a)** Cost-rate scatter: DTV/self-refine occupies the upper-left region (low cost, high pass rate) on both tasks. **(b)** Cumulative pass rate vs. per-case cost: DTV exhibits the early-rise effect on both tasks and a continued-rise pattern on JS to TS that distinguishes it from naïve's plateau (Section 4.1).

J.2 Pass rate breakdown

Table 5 gives the full pass-rate and token-cost breakdown underlying the compact RQ1 summary in Section 4.1.

Table 5: Pass rate, average tokens-per-case ratio k, and tokens-per-pass (mean over passing cases, with 95% bootstrap CI) for all four configurations on both tasks. Per-case paired token-cost statistics on the both-pass subset are reported separately in Appendix M.6.

Task	**Configuration**	**Pass**	**Pass rate**	**Avg. k**	**Tokens / pass [95% CI]**
C → Rust (n =300)	naive/one-shot	49/300	16.3%	1.55	360 [309,415]
	naive/self-refine	217/300	72.3%	7.18	1903 [1548,2339]
	DTV/one-shot	141/300	47.0%	2.00	596 [499,714]
	DTV/self-refine	246/300	82.0%	5.28	1654 [1345,1990]
JS → TS (n =150)	naive/one-shot	7/150	4.7%	1.10	435 [310,567]
	naive/self-refine	50/150	33.3%	11.48	1648 [1071,2497]
	DTV/one-shot	32/150	21.3%	2.77	715 [547,960]
	DTV/self-refine	69/150	46.0%	10.11	2103 [1470,2856]

J.3 Runtime and verifier invocations

Table 6 reports recorded per-case elapsed time for the matched self-refine configurations, starting after model loading and source preparation and ending before output persistence. Table 7 separates outer verification rounds from DTV's inner verification calls. An inner call is a boundary-level verification event; on JS to TS it invokes the compiler and linter serially. These counts are not interchangeable with the controller's action count. In-loop repair trades more intermediate checks for fewer outer attempts and less continuation of failing prefixes.

Table 6: Pass rate and mean elapsed time under the shared $k = 16$ cap. The task-specific cohorts are shown explicitly. Time includes generation and verification.

Model	Task	n	Pass rate (%)		Seconds/case		Time ratio
			naïve	DTV	naïve	DTV	DTV/naïve
Qwen3-4B	C-to-Rust	300	72.3%	82.0%	245.1	192.8	0.79
Qwen3-4B	JS-to-TS	150	33.3%	46.0%	588.7	1,091.4	1.85
Qwen3-30B-A3B	C-to-Rust	100	92%	97%	173.3	98.1	0.57
Qwen3-30B-A3B	JS-to-TS	100	57%	66%	807.9	1,017.9	1.26

Table 7: Mean outer verification rounds and DTV inner calls per case for the cohorts in Table 6.

Model	Task	Outer rounds/case		DTV inner calls/case
		naïve	DTV	
Qwen3-4B	C-to-Rust	5.49	3.08	208.5
Qwen3-4B	JS-to-TS	9.27	8.23	259.6
Qwen3-30B-A3B	C-to-Rust	2.34	1.34	82.11
Qwen3-30B-A3B	JS-to-TS	6.82	5.02	363.83

The measured mean latency is 0.054 seconds for a native `rustc` check and 1.823 seconds for the serial TypeScript/ESLint pair, which starts separate Node.js processes. These are CPU-side costs, whereas model decoding uses the GPU (hardware in Appendix F). The Qwen3-30B-A3B experiment reduces the DTV/naïve elapsed ratio on both tasks while retaining higher pass rates. Since the TypeScript and ESLint checks are independent, parallel execution would reduce the pair latency toward the slower check, approximately 1.253 seconds. This is a latency estimate; Table 6 reports the measured serial implementation.

Takeaway. DTV turns its token-budgeted success gains into wall-clock savings when verifier calls are inexpensive. With Qwen3-30B-A3B, its relative time savings grow on C to Rust and its latency premium narrows on JS to TS, while retaining higher pass rates on both tasks. This lower relative overhead strengthens the practical advantage of in-loop verification with a stronger generator.

J.4 Shared strict TypeScript recheck

The inner diagnostic filter affects DTV's search, while every method's completed output faces the same unfiltered final compiler/linter gate (Appendix E). We additionally recheck the saved JS-to-TS outputs with TypeScript 5.9.3 under `strict: true`, holding the setting identical across methods. Compiler options are `noEmit: true`, `skipLibCheck: true`, target ES2020, and the ES2020 and DOM libraries, with the same installed type declarations. A strict success must satisfy both the original final criterion and this extra compiler check; the denominator remains the entire cohort.

Table 8 shows that DTV retains its pass-rate advantage across all evaluated generators, although the stricter criterion reduces absolute pass rates.

Table 8: JS-to-TS pass rates under the common final compiler/linter criterion and its conjunction with `tsc --strict`. All rows compare naïve/self-refine with DTV/self-refine.

Model	n	Shared final criterion		Final + strict	
		naïve	DTV	naïve	DTV
Qwen3-4B	150	33.3%	46.0%	20.7%	26.7%
Gemma-4-E4B	100	50.0%	58.0%	30.0%	36.0%
Qwen3-30B-A3B	100	57.0%	66.0%	34.0%	42.0%

Takeaway. DTV/self-refine yields more outputs that pass both the original final gate and the strict recheck than naïve/self-refine across all evaluated generators. It therefore delivers higher verified success under the same token budget even when completed translations must satisfy stronger type requirements.

J.5 Cost-matched baseline grid

Table 9 reports the full pass-rate and per-case token-cost breakdown for the cost-matched baseline grid summarized in Section 4.1. The grid compares the head-to-head configurations with best-of-N for $N \in \{1, 2, 4, 8, 16, 32\}$ and S* on the cost-matched subsets ($n = 200$ for C to Rust, $n = 100$ for JS to TS); naïve/self-refine and DTV/self-refine are repeated here at the subset n for direct comparison against the BoN and S* columns.

Table 9: Pass rate, average token cost per case, tokens per successful pass, and per-case token cost relative to DTV/self-refine for the cost-matched baseline grid on the n=200 C-to-Rust and n=100 JS-to-TS subsets. The $\times$ ratio is per-case tokens divided by DTV/self-refine's per-case tokens on the same task. naïve/self-refine and DTV/self-refine are repeated here at the subset n for direct comparison.

Task	Configuration	Pass	Pass rate	Tok/case	Tok/pass	vs DTV (tok/case)
C $\rightarrow$ Rust (n =200)	naive/self-refine	152/200	76.0%	5266	1798	1.51×
	DTV/self-refine	169/200	84.5%	3489	1805	1.00×
	naive/BoN-1	31/200	15.5%	737	377	0.21×
	naive/BoN-2	42/200	21.0%	1437	476	0.41×
	naive/BoN-4	58/200	29.0%	2772	928	0.79×
	naive/BoN-8	73/200	36.5%	5154	1880	1.48×
	naive/BoN-16	84/200	42.0%	9574	2573	2.74×
	naive/BoN-32	89/200	44.5%	17866	3445	5.12×
	S*/n8r3	151/200	75.5%	16563	11551	4.75×
JS $\rightarrow$ TS (n =100)	naive/self-refine	37/100	37.0%	15105	1740	1.09×
	DTV/self-refine	50/100	50.0%	13815	2185	1.00×
	naive/BoN-1	7/100	7.0%	1186	435	0.09×
	naive/BoN-2	7/100	7.0%	2377	435	0.17×
	naive/BoN-4	7/100	7.0%	4727	435	0.34×
	naive/BoN-8	7/100	7.0%	9509	435	0.69×
	naive/BoN-16	7/100	7.0%	18871	435	1.37×
	naive/BoN-32	7/100	7.0%	37664	435	2.73×
	S*/n8r3	33/100	33.0%	27388	9751	1.98×

J.6 Paired comparisons (compile pass)

Table 10 reports the paired compile-pass comparisons used to separate DTV's gains from case-level difficulty variation.

Table 10: Paired McNemar tests on per-case compile pass for the four canonical comparisons on both tasks. The Only-A and Only-B columns report the discordant counts. All comparisons in the main text are paired (same case identifiers under the same dataset and budget).

Task	Baseline (A)	DTV (B)	n	Only A	Only B	Diff (pp)	McNemar p
C $\rightarrow$ Rust	naive/one-shot	DTV/one-shot	300	13	105	+30.7	< 0.001
	naive/self-refine	DTV/self-refine	300	21	50	+9.7	< 0.001
JS $\rightarrow$ TS	naive/one-shot	DTV/one-shot	150	1	26	+16.7	< 0.001
	naive/self-refine	DTV/self-refine	150	7	26	+12.7	0.001

J.7 Functional guardrail (C to Rust)

The pass-rate metric in the main body uses compilation as the primary criterion because compilation is the dense in-loop verifier signal that the DTV controller consumes. To rule out the possibility that DTV's compile-rate gains are achieved at the expense of behavioral correctness (for example, by patching function bodies with stubs that compile but fail differential tests), we run a paired functional analysis on the matched outer-loop setting: naïve/self-refine versus DTV/self-refine. A case is a full success only if the Rust translation compiles and matches the C source on all differential test inputs; the per-case test pass rate is the fraction of differential tests passed and is set to zero on compilation failure. Table 11 reports the marginal outcomes per configuration, with the conditional column giving full-success rate among cases that compile.

The marginal view shows DTV's conditional full-success rate (10.2%) below naïve's (13.0%), but this conditional comparison conditions on different sets of cases: DTV compiles 246 of 300 cases versus naïve's 217, so DTV's denominator includes harder cases that naïve cannot compile at all. The cleanest way to remove this selection effect is to restrict to the cases where both configurations produce a compiling translation. On this matched compile set (Table 12, $n = 195$), DTV's full-success rate is 10.8% versus naïve's 12.3% (-1.54 pp; McNemar $p = 0.549$), and its average per-case

Table 11: Marginal functional outcomes for naïve/self-refine and DTV/self-refine on C to Rust ($n = 300$). Compile is the share of cases whose Rust translation compiles, the average test pass rate is the per-case fraction of differential tests passed (zero on compilation failure) averaged over all n cases, and the conditional column gives the full-success rate among the cases that compile.

Configuration	Compile	Avg. test pass rate	Full | compile
naïve/self-refine	217/300 (72.3%)	23.9%	13.0%
DTV/self-refine	246/300 (82.0%)	22.4%	10.2%

differential test rate is 29.6% versus naïve's 32.8% (-3.18 pp, 95% bootstrap CI $[-7.49, +1.14]$). Neither difference reaches significance and the bootstrap CI straddles zero, so on cases where both methods produce a compiling target, DTV's behavioral correctness is statistically indistinguishable from naïve's.

Table 12: Both-compile paired comparison on C to Rust, restricted to cases where both naïve/self-refine and DTV/self-refine produce a compiling translation ($n = 195$). The full-success columns report the share of these cases that pass all differential tests; the average rate column averages the per-case fraction of differential tests passed, with a 95% paired-bootstrap CI on the difference.

n	naïve full	DTV full	Full diff (pp)	McNemar	Avg. rate diff [95% CI]
195	12.3%	10.8%	-1.54	$p = 0.549$	-3.18 pp $[-7.49, +1.14]$

The full-sample paired view (Table 13) confirms this conclusion across all 300 cases. The discordant counts are 11 (only naïve) versus 8 (only DTV), giving a difference of -1.00 pp on full success (McNemar $p = 0.648$); the paired test pass rate difference is -1.5 pp with a 95% bootstrap CI of $[-5.2, +2.3]$ pp. Together, the both-compile and full-sample paired analyses rule out a hidden behavioral regression: DTV's compile-rate gains are not paired with worse differential correctness.

Table 13: Full-sample paired comparison on full-functional success for naïve/self-refine versus DTV/self-refine across all 300 C-to-Rust cases. "Only naïve" and "Only DTV" count cases where only the named method achieves full success. The test rate diff is the paired difference in average test pass rate (in percentage points) with a 95% bootstrap CI.

Only naïve	Only DTV	Diff (pp)	McNemar	Test rate diff (pp) [95% CI]
11	8	-1.00	$p = 0.648$	-1.5 $[-5.2, +2.3]$

K Cross-model robustness

This appendix presents the cross-model analysis referenced in Section 4.1. Figure 7 compares Qwen3 and Gemma4 on matched task subsets; Figure 8 adds Qwen3-30B-A3B on the first 100 cases per task. The metric hierarchy remains pass rate and token cost, with C-to-Rust differential testing as a guardrail. In the Qwen3-4B/Gemma comparison, DTV exceeds naïve at the budget cap on three pairs (+8.5 pp, +13.0 pp, +8.0 pp) and reduces normalized cost on three (-23.6%, -11.5%, -14.5%). Gemma C to Rust exhibits the budget-dependent crossover discussed below.

On Gemma4 C to Rust, the comparison is budget-dependent (Figure 7, left panel). DTV/self-refine reaches a higher pass rate than naïve/self-refine across the lower-to-mid budget range, with a peak lead of $+10.5$ pp at $k \approx 4$; the curves cross near $k \approx 7$ and naïve/self-refine continues to climb past DTV through the cap, ending at 93.5% versus 86.0%. We attribute the erosion of DTV's early-rise advantage to the high baseline pass rate on this pair: with naïve/self-refine already passing 93.5% of cases at the cap, the residual hard cases provide fewer opportunities for DTV's verify-and-rollback to convert into marginal passes than naïve's outer-loop retry produces over the same total budget. The per-case failure modes that drive the cap-budget gap, and whether they fall outside the scope of DTV's structure-aware rollback, are part of the per-case analysis in Section 4.3. Behavioral correctness on the differential-testing guardrail is unchanged on this pair (15.00% vs 15.00% in functional pass rate; Appendix K.4).

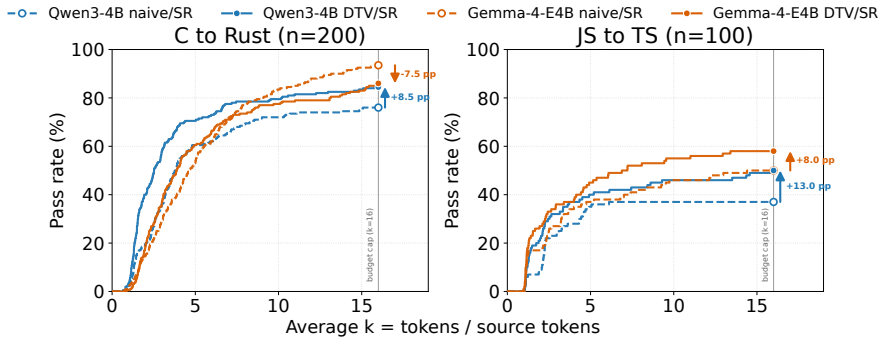


Figure 7: Cross-model cumulative pass rate vs. normalized per-case cost k for naïve/self-refine and DTV/self-refine on both tasks. DTV exceeds naïve across the lower budget range on all four pairs; at the budget cap, DTV remains ahead on three pairs, while on Gemma4 C to Rust the curves cross near $k \approx 7$ and naïve overtakes (behavioral correctness preserved; Appendix K.4).

On JS to TS, both model families show DTV pass-rate gains at the budget cap (+13.0 pp on Qwen3, +8.0 pp on Gemma4) at reduced normalized cost (-11.5% and -14.5% respectively). Across the four pairs, DTV's gains on both pass rate and per-case cost generalize across model families: on three pairs DTV exceeds naïve at the budget cap in both metrics, and across the lower portion of the budget range DTV exceeds naïve in pass rate on all four. The single cap-budget exception is concentrated on the pair where the baseline is highest (Gemma4 C to Rust at 93.5%).

K.1 Qwen3-30B-A3B scaling and one-shot decomposition

We evaluate `Qwen/Qwen3-30B-A3B-Instruct-2507`, a mixture-of-experts generator with 30.5B total parameters, on the first 100 cases of each randomly sampled main-evaluation cohort. These are prefixes of the same pools used above, rather than replacements for the 200-case Gemma C-to-Rust comparison. Both methods use the same $k = 16$ cap, prompts, and final verifiers. Figure 8 reconstructs cumulative pass rates from the raw batch records, placing each step at a successful case's actual generated-token cost normalized by its source-token count. The curves use full-budget runs rather than independent reruns at each cutoff. Crash and missing-output cases count as failures at every cutoff, including two JS-to-TS DTV crash records and one missing case.

At the integer cutoffs from $k = 2$ through $k = 16$, DTV leads on C to Rust, ending at 97% versus 92%. On JS to TS it leads at the integer cutoffs from $k = 1$ through $k = 16$ and ends at 66% versus 57%. The low-budget advantages are pronounced at $k = 2$: 87% versus 70%, and 45% versus 18%.

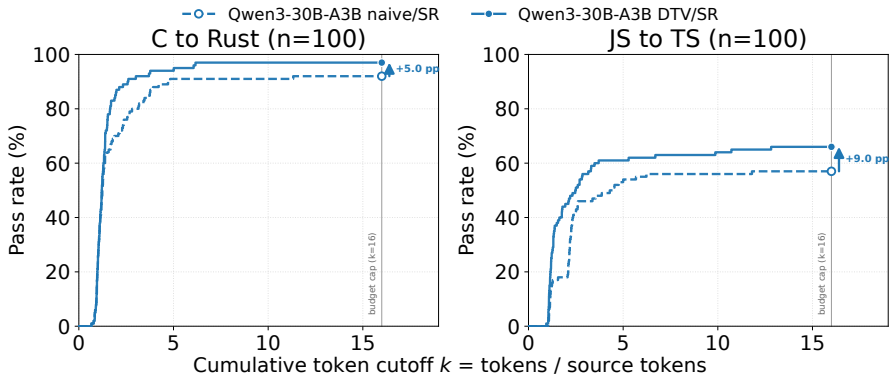


Figure 8: Qwen3-30B-A3B cumulative pass rates versus normalized per-case cost, reconstructed from raw $k = 16$ batch records on 100 cases per task. Steps mark actual successful-case costs. Dashed: naïve/self-refine; solid: DTV/self-refine.

Table 14 isolates the first complete-program attempt, excluding outer self-refinement. DTV improves its pass rate from 67% to 89% on C to Rust and from 18% to 48% on JS to TS. The reported mean k values condition on cases with a recorded first outer verification. One-shot is an operating point defined by the first attempt, not by the CDF cutoff $k = 1$; a first attempt may itself consume more than one source-length of tokens.

Table 14: First-attempt Qwen3-30B-A3B results. Pass rates use the full 100-case cohorts; mean normalized cost k uses cases with a recorded first outer verification.

Task	Configuration	Pass / cases	Mean k
C to Rust	naïve/one-shot	67/100	1.13
C to Rust	DTV/one-shot	89/100	1.25
JS to TS	naïve/one-shot	18/100	1.13
JS to TS	DTV/one-shot	48/100	1.62

Takeaway. With Qwen3-30B-A3B, DTV yields more successful translations than the naïve baseline both on the first complete-program attempt and at the shared token-budget cap. It therefore improves a stronger generator's first-attempt reliability as well as its success under a fixed overall budget, demonstrating benefits beyond additional outer retries.

K.2 Marginal stats

Table 15 reports per-cell pass rate, average per-case k, and tokens per pass for naïve/self-refine and DTV/self-refine on both tasks. Each Gemma cell uses the corresponding RQ1 sample list ($n = 200$ for C to Rust, $n = 100$ for JS to TS); Qwen cells are restricted to the same prefixes for paired cross-model comparison, so the matched case set within each (model, task) cell has the same case identifiers across configurations.

Table 15: Per-cell pass rate, average per-case k, and tokens per pass for naïve/self-refine and DTV/self-refine on both tasks. k is normalized in each model's own tokenizer. Costs use recorded token consumption for both passing and failing cases; missing cases are assigned $k = 16$.

Cell	**Strategy**	**Pass**	**Pass rate**	**Avg. k**	**Tokens / pass**
Qwen3-4B × C → Rust (n =200)	naive/self-refine	152/200	76.0%	6.77	1798
	DTV/self-refine	169/200	84.5%	5.17	1805
Gemma-4-E4B × C → Rust (n =200)	naive/self-refine	187/200	93.5%	5.91	3327
	DTV/self-refine	172/200	86.0%	5.99	3024
Qwen3-4B × JS → TS (n =100)	naive/self-refine	37/100	37.0%	10.96	1740
	DTV/self-refine	50/100	50.0%	9.70	2185
Gemma-4-E4B × JS → TS (n =100)	naive/self-refine	50/100	50.0%	9.87	3515
	DTV/self-refine	58/100	58.0%	8.43	3692

K.3 Paired comparisons

Within each cell, both methods see the same case identifiers under the same per-case budget cap, so paired McNemar tests on per-case pass directly isolate method-attributable gain from case-level difficulty variation (Table 16). To separate per-success efficiency from marginal pass-rate composition differences, Table 17 restricts to the subset of cases where both methods produce a passing translation and compares average k on that matched both-pass subset.

K.4 Functional guardrail (C to Rust)

Mirroring RQ1's functional guardrail (Appendix J.7), Table 18 reports differential-testing outcomes on both C-to-Rust cells to check whether DTV's pass-rate change accompanies a behavioral-correctness loss. The Qwen functional values here use the Gemma-matched 200 cases; the full-300 Qwen view is in Appendix J.7.

Table 16: Paired McNemar tests on per-case pass within each cell (same case identifiers under naïve vs. DTV). Discordant counts: "Only naïve" = naïve passes and DTV does not; "Only DTV" = DTV passes and naïve does not.

Cell	n	Only naive	Only DTV	Diff (pp)	McNemar p
Qwen3-4B × C → Rust	200	15	32	+8.5	0.019
Gemma-4-E4B × C → Rust	200	19	4	-7.5	0.003
Qwen3-4B × JS → TS	100	6	19	+13.0	0.015
Gemma-4-E4B × JS → TS	100	11	19	+8.0	0.200

Table 17: Both-pass subset analysis: average k on the cases where both naïve/self-refine and DTV/self-refine produce a passing translation. This view isolates DTV's per-success efficiency from the marginal pass-rate composition difference reported in Table 15. Δk is the percentage difference (DTV minus naïve), normalized by naïve's mean k; negative values favor DTV.

Cell	Both-pass n	naive avg. k	DTV avg. k	Δk (%)
Qwen3-4B × C → Rust	137	3.54	2.95	-16.5%
Gemma-4-E4B × C → Rust	168	5.10	4.66	-8.7%
Qwen3-4B × JS → TS	31	2.54	3.39	+33.3%
Gemma-4-E4B × JS → TS	39	3.01	2.49	-17.2%

On the matched 200 ids, naïve/self-refine on Qwen3 produces 11.50% full-functional success and DTV/self-refine 9.50% (-2.0 pp; McNemar $p = 0.424$). The marginal difference is small in absolute terms (4 cases out of 200) and consistent with the n.s. RQ1 result on the full 300 set; we therefore do not interpret it as a substantive behavioral-correctness loss.

On Gemma4 C to Rust, the marginal full-functional success rate is identical between naïve/self-refine and DTV/self-refine (15.00% versus 15.00%, exact tie; McNemar $p = 1.000$). DTV's compiling pool is smaller than naïve's by 15 cases (170 versus 185), but the conditional full-success rate within DTV's compiling pool is higher than naïve's (17.65% versus 16.22%), suggesting that the cases DTV does compile are more likely to also pass differential tests. Aggregating both views, DTV's pass-rate regression on this near-saturation cell does not coincide with a behavioral-correctness regression; on the cases DTV does compile, behavioral pass-through is in fact slightly higher than naïve's, indicating that the lost compiles concentrate among translations whose compile-pass-but-test-fail status would not have been detectable by the in-loop oracle alone.

L RQ2 detailed results

This appendix complements RQ2 (Section 4.2) with ablation costs, paired tests, outer-loop rescue accounting, and escalation traces.

L.1 Cost breakdown

Table 19 reports the full per-configuration cost breakdown underlying the compact RQ2 summary in Section 4.2. The Tokens/pass column reports the mean tokens consumed on cases that compile-pass with a 95% bootstrap CI, mirroring the reporting convention of Appendix J.2. DTV-full's per-success cost (1694) is higher than each of the three ablations' (1412, 1437, 1173); the gap reflects the additional in-loop verification and structured-recovery work DTV-full performs on cases that succeed. At the per-case granularity at which the budget is actually allocated, DTV-full nevertheless converts more of the budget into passing translations (the Avg. tokens column) than two of the three ablations.

L.2 Paired ablation comparisons

Table 20 reports paired McNemar tests for each ablation against the DTV-full baseline on three metrics. Within each row, both configurations see the same case identifiers, so the discordant counts (Only-A: baseline-only pass; Only-B: ablation-only pass) directly attribute outcome differences to the ablation rather than to case-level difficulty variation.

Table 18: Functional outcomes for naïve/self-refine and DTV/self-refine on C to Rust under the matched cross-model case set ($n = 200$ per model). Same metrics as RQ1's functional guardrail (Appendix J.7); the Qwen functional values here differ from the RQ1 appendix because RQ1 uses the full $n = 300$ set while RQ2 restricts to the Gemma-matched first 200.

Cell	Strategy	Compile	Full success	Full | compile	McNemar p (full)
Qwen3-4B × C → Rust	naive/self-refine	151/200	23/200 (11.50%)	23/151 (15.23%)	0.424
	DTV/self-refine	168/200	19/200 (9.50%)	19/168 (11.31%)	
Gemma-4-E4B × C → Rust	naive/self-refine	185/200	30/200 (15.00%)	30/185 (16.22%)	1.000
	DTV/self-refine	170/200	30/200 (15.00%)	30/170 (17.65%)	

Table 19: Per-configuration pass rate, average tokens per case, and Tokens/pass with 95% bootstrap CI on the first 100 C-to-Rust cases. The Avg. tokens column is the per-case mean over all cases (failures hit the budget cap); the Tokens/pass column averages tokens over compile-passing cases, with a bootstrap CI on this mean.

Configuration	Pass rate	Avg. tokens	Tokens / pass [95% CI]
DTV-full	85/100 (85.0%)	3580	1694 [1164,2405]
DTV-no-feedback	74/100 (74.0%)	4138	1412 [1157,1691]
DTV-no-escalation	79/100 (79.0%)	3623	1437 [1025,2000]
DTV-detect-and-abort	70/100 (70.0%)	4417	1173 [937,1416]

The compile-rate drops for detect-and-abort (-15.0 pp, $p = 0.003$) and no-feedback (-11.0 pp, $p = 0.035$) are paired-significant. The no-escalation drop (-6.0 pp, $p = 0.263$) is directionally consistent with the other two but falls below the detectability floor at $n = 100$. Inner-1shot drops for detect-and-abort (-30.0 pp) and no-feedback (-23.0 pp) are large enough to be detected at this sample size ($p < 0.001$ and $p < 0.001$ respectively); no-escalation leaves inner-1shot unchanged ($p = 1.000$). All three functional differences fall below the detectability floor and should be read alongside the descriptive ordering reported in Section 4.2.

L.3 Outer self-refine rescue contribution

Outer self-refine compensates for inner-loop ablations by rerunning DTV with accumulated diagnostic feedback in the prompt. Table 21 reports the rescue rate (cases recovered by outer retry as a fraction of inner-1shot failures) for each configuration. Rescue rates fall in a 62–73% band across all four configurations, indicating outer self-refine's rescue capacity is roughly comparable across inner-loop ablations. Consequently, the gap between inner-1shot and final compile rates (Table 21) reflects what outer self-refine can compensate for rather than how the ablation interacts with the outer loop.

L.4 Trace walk-throughs: when escalation helps and hurts

We examine two paired cases illustrating the mechanism underlying the -6.0 pp final-pass-rate gap between DTV-full and no-escalation. The runs use temperature sampling rather than greedy decoding, so paired-trace differences are mechanistic rather than deterministic counterfactual.

Escalation rescues a non-local mutability error. Case `s368372837` (only-A: DTV-full passes, no-escalation fails) is a non-local error: rustc reports `cannot assign twice to immutable variable sold` at a mutation site and suggests making an earlier binding mutable. DTV-full attempts three statement-scope rollbacks to the same prefix length, each producing the same diagnostic; the statement-stall counter then triggers escalation to block scope, which truncates the committed prefix back through the binding region. Regeneration produces a passing verify and commit, and the case finishes in 652 generated tokens. No-escalation cycles through statement repair until the per-case budget cap of 3600 tokens, without reaching outer-loop verification. Across the no-escalation cohort, 10 cases exhaust their budget within inner-loop activity (zero recorded outer rounds), consistent with statement-only rollback failing to make progress on these diagnostics.

Escalation discards a useful prefix. Case `s126370263` (only-B: no-escalation passes, DTV-full fails) shows the opposite failure mode. DTV-full repeatedly attempts patch-based repair on a local

Table 20: Paired McNemar tests on per-case compile, inner-1shot, and functional pass for each ablation against the DTV-full baseline ($n = 100$ paired cases). Only-A: baseline-only pass; Only-B: ablation-only pass. With $n = 100$, McNemar can reliably detect effects of approximately 10 pp or larger; differences below this floor with non-significant p-values should be read as point estimates rather than null effects.

Comparison	Metric	Only-A	Only-B	n	Diff (pp)	McNemar p
DTV-full vs DTV-no-feedback	Compile (HEADLINE)	17	6	100	-11.0	0.035
DTV-full vs DTV-no-feedback	Inner-1shot	29	6	100	-23.0	< 0.001
DTV-full vs DTV-no-feedback	Functional (guardrail)	6	4	100	-2.0	0.754
DTV-full vs DTV-no-escalation	Compile (HEADLINE)	13	7	100	-6.0	0.263
DTV-full vs DTV-no-escalation	Inner-1shot	17	17	100	+0.0	1.000
DTV-full vs DTV-no-escalation	Functional (guardrail)	7	2	100	-5.0	0.180
DTV-full vs DTV-detect-and-abort	Compile (HEADLINE)	19	4	100	-15.0	0.003
DTV-full vs DTV-detect-and-abort	Inner-1shot	34	4	100	-30.0	< 0.001
DTV-full vs DTV-detect-and-abort	Functional (guardrail)	4	2	100	-2.0	0.688

Table 21: Outer self-refine rescue accounting: rescued cases (final compile − inner-1shot compile) as a fraction of inner-1shot failures, for each configuration. The rescue rate band (62.5–73.2%) indicates that outer self-refine's rescue capacity is roughly independent of which inner mechanism is ablated.

Configuration	Inner-1shot	Final compile	Rescued	Rescue rate
DTV-full	44/100 (44.0%)	85/100 (85.0%)	41/56	73.2%
DTV-no-feedback	21/100 (21.0%)	74/100 (74.0%)	53/79	67.1%
DTV-no-escalation	44/100 (44.0%)	79/100 (79.0%)	35/56	62.5%
DTV-detect-and-abort	14/100 (14.0%)	70/100 (70.0%)	56/86	65.1%

`if/else` expression; after several stalls the controller escalates to function-scope rollback, which truncates the committed prefix to length zero and forces regeneration from scratch. The regenerated trajectory does not converge within the budget. No-escalation on the same case applies a single statement-scope rollback to fix an unrelated macro syntax error, commits, and passes on the first outer round.

Aggregate caveat. These two cases illustrate the canonical positive and negative interactions between rollback scope and the regeneration trajectory, but they do not generalize mechanically to all 20 paired identity flips. Of the 13 only-A cases, DTV-full uses non-statement rollback on 7; the remaining 6 succeed without scope widening in DTV-full and fail under no-escalation, indicating that escalation policy also affects generation trajectories indirectly (e.g., function-scope rollback resets the feedback mechanism from patch-based to inline continuation, exposing the model to a different repair-prompt format). Of the 7 only-B cases, 5 show DTV-full taking non-statement rollback (consistent with over-aggressive scope widening discarding useful prefix); the other 2 show no scope-widening at all, suggesting stochastic trajectory variation under temperature sampling.

M RQ3 detailed results

This appendix complements the main-body RQ3 results (Section 4.3) with the fix-shape classifier algorithm, the full fix-shape breakdown including UNKNOWN counts and the alternative R_{final} classification, the per-error-code rescue rate, the shared-code ratio distribution underlying the on-rescued mechanism claim, and the fix-shape transfer of the rescued cohort from one-shot to self-refine.

M.1 Fix-shape classifier algorithm

The fix-shape classifier introduced in Section 4.3 labels each naïve R1-failure case by the shape of the edit that naïve's R2 (its repair attempt) actually attempted, computed mechanically from the line-level diff between R1 and R2. No human labels enter the classification. These diagnostic-line overlap classes do not determine whether the edited region is reachable by block/function rollback. The upstream-edit analysis in Appendix N.2 supplies that complementary motivation for scope widening, using final passing programs as an endpoint proxy.

For each naïve case where round 1 fails, let R_1 denote naïve's first-attempt code and R_2 its second-attempt code. Let E denote the set of R_1 line numbers reported as errors in the diagnostic block fed back to round 2; for C to Rust the line numbers are extracted from `rustc`'s primary span markers (`- -> program.rs:L:C`), and for JS to TS from `tsc`'s and ESLint's primary error markers (`- LL:C: error:`). Let C denote the set of R_1 line numbers that R_2 modified, computed from `difflib.SequenceMatcher` opcodes: replace and delete chunks mark every R_1 line in the chunk; insert chunks mark the R_1 line immediately after the insertion point (or line 1 if the insertion is at the top). Let n_{R_1} denote the number of R_1 lines.

The four-class label and a sub-reason for transparency are assigned by Algorithm 2, with the first matching rule winning.

Algorithm 2 Fix-shape classifier (mode A). Given the modified line set C, the error line set E, and the R_1 length n_{R_1}, label the case as LOCAL, NONLOCAL, MIXED, or UNKNOWN.

Require: n_{R_1}, E, C
if $n_{R_1} = 0$ **then**
  return UNKNOWN {EXTRACT_FAIL}
else if $C = \emptyset$ **then**
  return UNKNOWN {NO_OP: R_2 produced no diff against R_1}
else if $|C|/n_{R_1} > 0.5$ **then**
  return UNKNOWN {REWRITE: R_2 rewrote the majority of R_1}
else if $E = \emptyset$ **then**
  return UNKNOWN {NO_ERROR_LINES: no extractable line numbers}
else if $C \setminus E = \emptyset$ **then**
  return LOCAL {every R_2 edit lands on an R_1 error line}
else if $C \cap E = \emptyset$ **then**
  return NONLOCAL {no R_2 edit lands on an error line}
else
  return MIXED {partial overlap}
end if

The 0.5 rewrite threshold is fixed across both tasks and across model families. UNKNOWN is reported separately and excluded from hierarchy claims because the fix shape is not determined by the diff alone in those sub-cases; the headline ordering in Section 4.3 (LOCAL $\geq$ MIXED $\geq$ NONLOCAL) is read from the three resolved classes.

The primary classification (mode A) uses R_2 as defined above. An alternative classification (mode B) uses R_{final}, the round of naïve's outer self-refine trace that ultimately compiled, in place of R_2, and is restricted to cases where naïve eventually passes; mode B is reported alongside mode A in Table 2. The two modes agree on 88.1% of mode-B-eligible cases on C to Rust (loose: 91.2%, n=159) and 84.1% on JS to TS (loose: 95.5%, n=44); the loose agreement collapses LOCAL and MIXED into a single ANCHORED bucket.

M.2 Fix-shape full breakdown

Table 22 reports the complete fix-shape rescue rate including the UNKNOWN bucket and the alternative R_{final} classification (mode B), complementing the slim main-body view in Table 2. The universe is naïve R1-failure cases (rescued plus both-fail; $n = 230$ on C to Rust under mode A, $n = 141$ on JS to TS under mode A). The UNKNOWN row sits at 57.9% on C to Rust ($n = 19$) and 20.0% on JS to TS ($n = 5$); we exclude UNKNOWN from the LOCAL $\geq$ MIXED $\geq$ NONLOCAL hierarchy claim because its constituent sub-reasons (no-op and rewrite) describe undetermined fix shapes rather than a coherent shape class.

The NONLOCAL bucket on JS to TS is underpowered. Mode A counts 5 NONLOCAL cases out of 141 R1-failure cases on JS to TS, with 2 cases entering the rescue-rate denominator; mode B counts 0 NONLOCAL cases ($n = 0$). Below the underpowered threshold of 5 cases, the rescue-rate point estimate is reported with a $*$ marker in Table 2 and is not used to claim a NONLOCAL gap on JS to TS in Section 4.3; the LOCAL versus MIXED gap on JS to TS (29.3% vs 11.9%, mode A) is well-powered ($n = 58$ and $n = 67$ respectively) and supports the LOCAL $\geq$ MIXED edge of the hierarchy on this task.

Table 22: Full fix-shape rescue rate on naïve R1-failure cases. Columns A use R_2 as the repair anchor (primary classification, matching the main-body Table 2); columns B use R_{final} from naïve-eventually-pass cases (alternative anchor). Cells report rescue rate (rate-eligible n); $*$ marks bins below the underpowered threshold of 5 cases.

Fix-shape	C → Rust A	C → Rust B	JS → TS A	JS → TS B
Local	50.0% (12)	100.0% (2*)	29.3% (58)	48.0% (25)
Mixed	43.5% (168)	50.4% (129)	11.9% (67)	37.5% (16)
Nonlocal	29.6% (27)	40.0% (10)	0.0% (2*)	– (0)
Unknown	57.9% (19)	75.0% (16)	20.0% (5)	0.0% (2*)

M.3 Per-error-code rescue rate

The fix-shape hierarchy in Section 4.3 categorizes cases by the shape of naïve's repair attempt; we additionally report per-error-code rescue rate as a complementary view organized by the diagnostic that broke naïve's R1 (Table 23). For each error code that appears on at least 5 naïve R1-failure cases, we report the number of such cases, the number where DTV/one-shot rescues, and the rescue rate. A case enters the denominator for every distinct error code that appears on its R1 diagnostic block.

Table 23: Per-error-code rescue rate on naïve R1-failure cases, top 6 codes by R1-failure prevalence on each task. A case may enter the denominator for multiple codes when its R1 diagnostic block reports several. Codes with fewer than 5 R1-failure cases are excluded.

Task	Code	Naive-fail n	DTV rescued	Rescue rate
C → Rust	`E0599`	79	43	54.4%
	`E0308`	66	22	33.3%
	`E0277`	61	24	39.3%
	`E0425`	49	15	30.6%
	`E0433`	14	4	28.6%
	`E0384`	8	4	50.0%
JS → TS	`@typescript-eslint/typedef`	118	23	19.5%
	`@typescript-eslint/explicit-function-return-type`	58	7	12.1%
	`TS2339`	43	1	2.3%
	`TS2554`	28	1	3.6%
	`TS1192`	21	0	0.0%
	`TS1259`	16	0	0.0%

On C to Rust, the highest rescue rates concentrate on borrow-check and type-mismatch families (`E0599` method resolution at 54.4%, `E0384` immutable-binding violations at 50.0%) where the diagnostic anchors a concrete fix on the error line. The lower-rescue families (`E0425` unresolved name at 30.6%, `E0433` unresolved trait at 28.6%) typically require a non-local edit at an import or declaration upstream of the error line. On JS to TS, the dominant code is `@typescript-eslint/typedef` (the missing-type-annotation rule under our zero-typedef policy) at 19.5%; the lower-rescue codes `TS2339`, `TS2554`, `TS1192`, `TS1259` are concentrated on cases where the correct annotation depends on cross-program usage rather than on a fact local to the variable declaration. The per-code ordering is consistent with the fix-shape hierarchy: codes whose canonical fix is local-cause and local-fix rescue at higher rates, and codes whose canonical fix lives upstream rescue at lower rates.

M.4 Shared-code edit-concentration ratio

Figure 9 reports the per-case distribution of the shared-code edit-concentration ratio used in the Section 4.3 mechanism analysis on rescued cases. The universe is the subset of rescued cases where DTV's inner verifier fired and structure-aware rollback intervened (DTV/one-shot passes after at least one inner-loop rollback), intersected with the shared-code subset (DTV inner-loop error codes and naïve R1 error codes have non-empty intersection). For each case in this universe, the ratio is computed as the number of distinct DTV inner-loop diagnostic emissions on the shared codes divided by the number of distinct naïve R1 diagnostic emissions on the same codes; a ratio below 1 indicates DTV emits fewer per-shared-code verify diagnostics than naïve accumulated as R1 errors on the same codes.

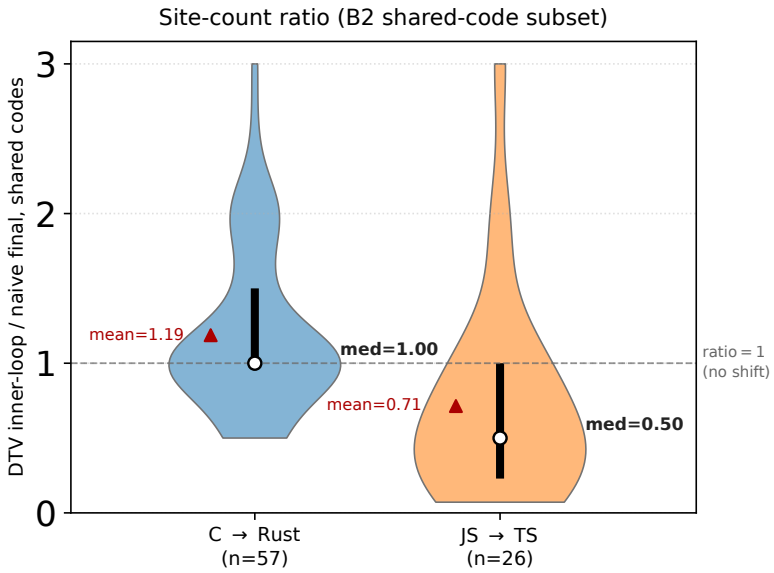


Figure 9: Per-case distribution of the shared-code edit-concentration ratio on rescued cases where DTV's inner verifier fired and the DTV/naïve code sets intersect. $n = 57$ on C to Rust and $n = 26$ on JS to TS. Ratios at or below 1 indicate DTV reaches rescue with at most as many per-shared-code verify-and-rollback cycles as naïve accumulated as R1 errors on the same codes.

The C-to-Rust distribution is centered at parity (median 1.00, with first and third quartiles Q_1=1.00 and Q_3=1.50, mean 1.19); the JS-to-TS distribution is shifted left of parity (median 0.50, Q_1=0.23, Q_3=1.00, mean 0.71). Both distributions sit at or below 1 at the median, supporting the Section 4.3 claim that DTV's incremental work is concentrated on the same error codes naïve's R1 hit rather than redirected to a different error class. The complementary both-fail comparison reinforces this view: 40.4% of C-to-Rust both-fail cases ($n = 59$ of 146) and 86.3% of JS-to-TS both-fail cases ($n = 101$ of 117) have DTV's inner-loop diagnostics intersecting naïve's R1 diagnostics, indicating that when DTV does not pass, it is most often stuck on the same error codes that broke naïve.

M.5 Paired round-of-pass statistics (self-refine)

Table 24 reports the paired round-of-pass statistics summarized in the Section 4.3 round-savings claim. The both-pass subset includes only cases where both naïve/self-refine and DTV/self-refine reach pass on the matched case identifiers. The med ratio and mean ratio columns are over the per-case naïve_rounds / DTV_rounds; the e/s/l column reports the count of cases where DTV reaches pass earlier, in the same outer round, or later than naïve; Δ is the per-case difference (naïve $-$ DTV) in outer rounds; sign p uses the two-sided exact sign test on the discordant pairs. The right-shifted ratio distributions (Q_1=1.00, Q_3=3.00on C to Rust; Q_1=1.00, Q_3=2.00on JS to TS) confirm the round-savings effect is broad rather than driven by tail outliers; both sign tests reject equality and both bootstrap CIs on the mean ratio exclude 1.0 (Table 24).

Table 24: Paired round-count statistics on the both-pass subset (matched naïve/self-refine and DTV/self-refine, same case identifiers). The e/s/l column counts cases where DTV passes in an earlier, the same, or a later outer round than naïve.

Task	n	med ratio	mean ratio [95% CI]	e/s/l	Δ mean	Δ [95% CI]	Sign p
C $\rightarrow$ Rust	196	2.00	2.06 [1.88, 2.27]	129/46/21	+1.22	[+0.94, +1.52]	$< 10^{-19}$
JS $\rightarrow$ TS	43	1.50	1.61 [1.38, 1.88]	24/15/4	+0.63	[+0.35, +0.93]	< 0.001

M.6 Both-pass token-cost analysis (self-refine)

The compile-pass paired analysis in Appendix J.6 establishes that DTV improves the binary outcome (more cases compile). This subsection addresses the orthogonal question used in the Section 4.3 round-savings analysis: on the cases where both methods succeed, does DTV use more or fewer tokens? Restricted to the both-pass subset of the matched naïve/self-refine versus DTV/self-refine comparison, Table 25 reports the per-case token-difference distribution (Δtok = DTV $-$ naïve in tokens; negative favors DTV).

Table 25: Per-case token-difference distribution on the both-pass subsets of the matched naïve/self-refine versus DTV/self-refine comparison (Δtok = DTV tokens $-$ naïve tokens; negative favors DTV). The DTV $<$ / $=$ / $>$ naïve columns report the number of cases where DTV uses fewer, equal, or more tokens (the equal column captures cases where both methods pass on first attempt with identical token counts). The trimmed mean drops the top 10% and bottom 10% of cases (the largest DTV-better and DTV-worse outliers symmetrically) to expose tail-driven shifts in the arithmetic mean.

Task	n	DTV $<$ naive	DTV $=$ naive	DTV $>$ naive	Median Δtok	Mean Δtok	Trimmed mean
C $\rightarrow$ Rust	196	133 (68%)	7 (4%)	56 (29%)	-208	-519	-297
JS $\rightarrow$ TS	43	24 (56%)	5 (12%)	14 (33%)	-61	+220	-109

Both-pass medians favor DTV on both tasks. On C to Rust ($n = 196$), DTV uses fewer tokens than naïve on 133 of the both-pass cases (67.9%), naïve uses fewer on 56 (28.6%), and the median paired difference is -208 tokens (sign test $p < 0.001$). On JS to TS ($n = 43$), DTV is the winner on 24 cases (55.8%) versus 14 for naïve (32.6%), with a median paired difference of -61 tokens (sign test $p = 0.143$, underpowered at this sample size). Both medians point in the same direction: on the typical case where both methods pass, DTV uses fewer tokens.

The mean disagrees with the median on JS to TS, driven by an asymmetric tail. On C to Rust the arithmetic mean (-519 tokens) agrees with the median direction; on JS to TS the mean is +220 tokens, opposite of the median. Inspecting the tails explains the discrepancy. Among the JS-to-TS both-pass cases, the top 4 (10%) where DTV uses most extra tokens contribute a cumulative +20875 tokens, while the bottom 4 where DTV saves most contribute only -7604 tokens (a 2.75$\times$ asymmetry favoring the DTV-bad tail). After symmetrically trimming 4 cases from each tail, the mean over the remaining 35 cases swings to -109 tokens, recovering the median direction. The C-to-Rust both-pass distribution has the opposite tail shape: top-10% DTV-good cases cumulatively save -98903 tokens versus top-10% DTV-bad cases costing +44106 tokens (0.45$\times$ in the opposite direction), so the C-to-Rust mean is robust to trimming and stays negative (-297 tokens after the same symmetric 10%/10% trim).

Mechanism behind the JS-to-TS DTV-bad tail. The structure-aware rollback mechanism underlying this asymmetric tail is described in Section 4.3: it ties back to the same non-local fix-shape boundary that lowers DTV's per-case rescue rate.

N Diagnostic guidance and repair scope

We examine how to guide regeneration after a failure and how much earlier code to make editable. We use ROCODE [15] as a representative implementation of the mechanism-level baselines: failed-path suppression for regeneration and entropy-based statement selection for rollback. The analyses test whether diagnostic feedback improves local repair success and whether an uncertainty-selected statement exposes observed upstream edits.

N.1 Matched regeneration-signal comparison

We isolate the regeneration signal by replaying the 81 initial compiler failures among the first 100 C-to-Rust cases evaluated with Qwen3-4B. For each failure, both arms resume from the same statement checkpoint with identical source context, sampling settings, and generation budgets, and are checked at the next structural boundary.

The diagnostic arm uses inline continuation with the `rustc` diagnostic appended as a block comment, supplying information about the constraint to repair. The path-decay arm discourages failed continuations without supplying diagnostic text. It reuses ROCODE's released trie and position-dependent decay [15] with factor 0.9, followed by its top-k/top-p sampling logic.[3]

The outcome requires reaching the shared boundary, producing a renderable artifact, and passing the same in-loop `rustc` oracle. All pairs reach the boundary; an unavailable rendered artifact counts as failure. Table 26 reports local one-repair pass rates, and Table 27 reports the paired outcomes. Diagnostic conditioning improves pass rate by 17.3 pp (exact two-sided McNemar $p = 0.0066$; paired-bootstrap 95% CI $[6.2, 28.4]$ pp, 20,000 resamples, seed 20260726). This outcome measures local verifier admissibility, not complete-program correctness.

Table 26: One-repair verifier pass under a shared failure state, statement checkpoint, renderer, and oracle.

Regeneration signal	Pass / cases	Pass rate
DTV diagnostic	36/81	44.4%
ROCODE path decay	22/81	27.2%

Table 27: Paired outcomes for the matched regeneration comparison ($n = 81$).

DTV diagnostic	ROCODE path decay	Cases
Pass	Pass	17
Pass	Fail	19
Fail	Pass	5
Fail	Fail	40

N.2 Observed upstream edits and entropy localization

We examine the 67 cases in the initial-failure cohort whose full DTV trajectory eventually compiles. For every failure-time artifact, we compare its generated-code tokens with the final passing program using a Tree-sitter terminal-token diff that excludes comments and formatting. A case is positive if at least one failure contains an observed edited line L_e before the first error-level diagnostic's primary line L_d. This identifies 45/67 cases (67.2%). As in the fix-shape analysis of Appendix M.1, the successful-program diff is a proxy for repair location, not evidence that each edit is necessary or unique. The aggregate and its classifications were manually checked.

For each positive case, we select its earliest qualifying failure, yielding one observation per case (44 inner failures and one outer failure). We teacher-force the artifact under the original source-translation prompt without diagnostic feedback. Following ROCODE's ordering, base-2 full-vocabulary entropy is computed after temperature scaling and before path decay or top-k/top-p filtering. We select the global maximum-entropy generated token and its containing statement. Let L_H be the selected token's line: exact-line hits satisfy $L_H = L_e$, while a rollback *exposes* the observed edit if the selected statement starts at or before L_e.

Table 28 reports the all-token analysis and the variant excluding comments and whitespace. Both locate the exact edit line in 2/45 cases, expose the edit in 17/45, and leave it frozen in 28/45; median signed distance is +3 lines. The selected cohort tests access to observed upstream edits; it does not evaluate complete repair trajectories.

[3] ROCODE implementation: https://github.com/jiangxxxue/ROCODE (MIT license).

Table 28: Entropy localization at the earliest qualifying failure of each observed-upstream-edit case ($n = 45$). Distance is in generated-code lines; "Code only" excludes comments and whitespace. "Too late" means the selected statement starts after the observed edit line, so rollback leaves that edit frozen. "Exact line" cases are included in "Exposes edit".

Tokens included	Exact line	Exposes edit	Too late	Median $L_H - L_e$
All generated tokens	2/45	17/45	28/45	+3
Code only	2/45	17/45	28/45	+3

Takeaway. Diagnostic-conditioned regeneration improves local repair success over failed-path suppression, while structural escalation can reopen earlier code left inaccessible by a too-late statement checkpoint. Together, these mechanisms give DTV more directed regeneration and greater flexibility to repair earlier decisions.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and Section 1 describe DTV as a decoding-time verification framework for code translation, scoped to two language pairs and frozen open-weight generators, and the pass-rate and cost claims match the head-to-head and cost-matched results reported in Section 4.1 and Section 4.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section A discusses oracle requirements, non-local semantic errors, implementation and evaluation scope (model and language pairs), and verifier wall-clock cost.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper presents an algorithmic framework and empirical evaluation; it contains no formal theorems, lemmas, or proofs.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: Section 2.2 and Algorithm 1 specify the controller; Appendices E, F, and H give datasets, seeds, oracle configurations, and the repair policy with its code configuration entry point. Appendices N.1 and N.2 specify the mechanism comparisons. The source release is linked in Section 1.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The source release is linked in Section 1; the underlying CodeNet and TypeWeaver datasets and the Qwen3 and Gemma models are publicly accessible and cited. Additional table and figure export inputs are documented with the analysis scripts.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 3 and Appendices E and F specify dataset construction (with sampling seeds), evaluation subsets, baseline configurations, sampler defaults, per-round and global generation caps, and the per-case token budget.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Headline tokens-per-pass and paired comparisons report 95% bootstrap confidence intervals, and paired pass-rate and round-of-pass differences are tested via McNemar's exact test or the two-sided sign test (Appendix J.2, J.7, L.1, M.5).

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Appendix F reports evaluation and replay hardware and generation caps. Table 6 and Appendix J.3 report elapsed time, verifier invocations, and timing boundaries.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The work uses publicly released code datasets and open-weight models, involves no human subjects or private data, and preserves anonymity in submission.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: Section B discusses the broader impacts.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: We release only a decoding-time controller that wraps existing open-weight models and public benchmarks; no new pretrained model, generative dataset, or other high-misuse-risk asset is released.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: The datasets and models are credited, and toolchain components are identified. Appendix N.1 additionally gives ROCODE's implementation source and MIT license for the reused path-decay mechanism.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The DTV source release linked in Section 1 accompanies the algorithmic, dataset-construction, and configuration documentation in Section 2.2 and Appendices E, F, and H.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: The paper does not involve crowdsourcing or research with human subjects, so IRB review does not apply.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: Qwen3-4B-Instruct, Qwen3-30B-A3B-Instruct, and Gemma-4-E4B-it are frozen generators inside the controller (Section 3, Appendix F); GPT-5.4 seeds AFL++ test-input generation (Appendix E).

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.