Verifiers in the Loop: Decoding Time Verification for Code Translation

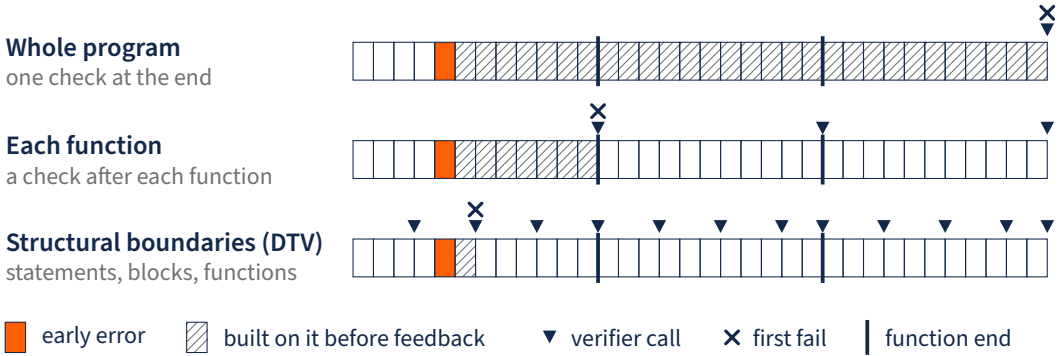
Tianyang Zhou[1], Somesh Jha[2,3],
Mihai Christodorescu[3], Kirill Levchenko[1],
Varun Chandrasekaran[1]

[1]University of Illinois Urbana-Champaign
[2]University of Wisconsin–Madison [3]Google
NeurIPS 2026

ILLINOIS

Verification timing is an algorithmic design choice

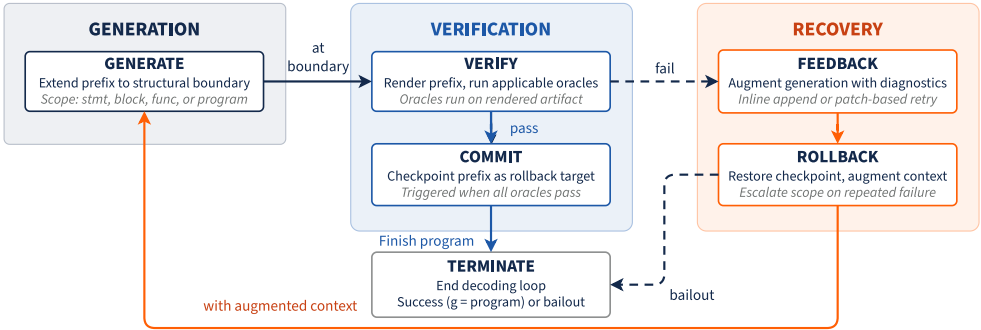
One early error, three designs: how much is built on it before the first feedback?



DTV puts verifiers inside the decoding loop

Verifying during decoding raises three questions: when to verify (Generate, Verify, Commit), how to roll back (Rollback), and what to do with the diagnostics (Feedback)

Verifier, as in the paper: a deterministic checker, not a proof. Oracles: rustc for C to Rust; tsc and ESLint for JS to TS



Verify at structural boundaries; commit only passing prefixes

Not after every token: most token prefixes are not well-formed program fragments. DTV verifies where a statement, block, function, or program ends

1. Verify: render the prefix, then run the oracles

```
fn sum(v: &[i32]) → i32 {
    let mut total = 0;
    for x in v {
        total += x;}
; todo!()}
```

The prefix ends at a statement. The renderer adds only the highlighted closing braces and a placeholder return value, `todo!()`

2. Commit: only if all applicable oracles pass

Commit: save the prefix as a checkpoint
Rollback returns to it, so it must be a *legal prefix*: one that can still be extended into a program that passes all oracles

✓ **All applicable oracles pass: commit**
Applicable: the oracle can run at this scope; here rustc, or tsc and ESLint

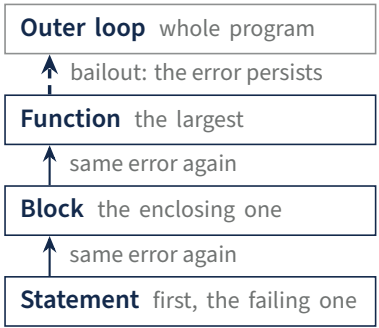
✕ **Any fails: no commit**
Recovery starts from the last checkpoint

How to roll back: start small, widen when the error repeats

Rollback should cut out the error but keep the correct code before it. The smallest scope keeps the most, but the cause can lie before the reported error

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

Retrying the failing `println!` cannot help; the fix is the earlier call, `consume(message.clone())`



What to do with diagnostics: feed them back to the model

Rolling back alone resamples from the same distribution, so the model can repeat the mistake. DTV adds the diagnostic to the context

Feedback A: Inline continuation

```
let arr: Vec<f64> = ... ;          // failed; rolled back
/* repair feedback:
   diagnostic: E0277 'Vec<f64>' cannot be built
               from an iterator over 'i32'
   hint: try changing 'f64' to 'i32'
*/
let arr: Vec<i32> = ... |          // model continues
```

Feedback B: Patch-based repair

```
let arr: Vec<f64> = ... ;          // failed; rolled back
[User message]
Failed snippet:  let arr: Vec<f64> = ... ;
Diagnostic:      E0277 expected 'i32', found 'f64'
Goal:            minimal patch at scope=stmt.

[Assistant response]
- let arr: Vec<f64> = ... ;
+ let arr: Vec<i32> = ... ;
```

Does the diagnostic help? 81 first rustc failures, one retry each from the same checkpoint

diagnostic added to the context (DTV)	**36/81 (44.4%)** pass
failed tokens made less likely, no diagnostic (ROCODE)	22/81 (27.2%) pass

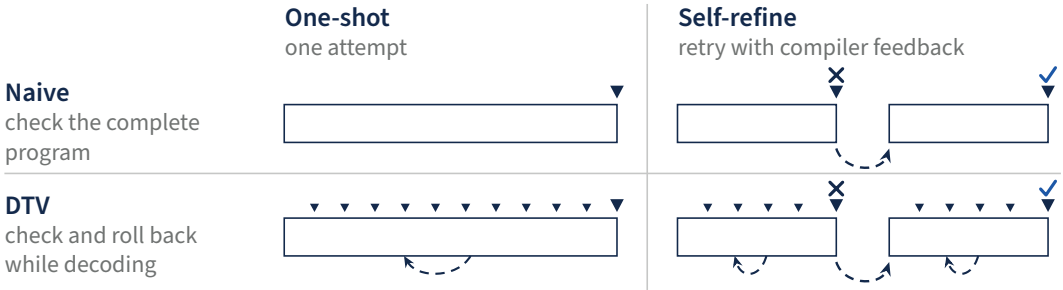
Qwen3-4B, first 100 C-to-Rust cases, same sampling and budget; paired McNemar p=0.0066

Setup: two tasks; two inner loops crossed with two outer loops

Qwen3-4B-Instruct on two tasks with different goals and verifiers

C to Rust	code translation	300 CodeNet programs	rustc
JS to TS	type annotation	150 TypeWeaver programs	tsc, ESLint

Inner loop: what happens inside one attempt. Outer loop: whether a failed program is retried

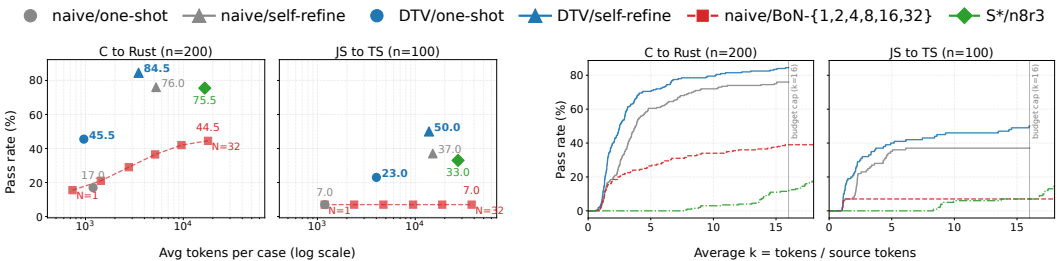


Higher pass rate per token than naive, best-of-N, and S*

Qwen3-4B-Instruct; 200 C-to-Rust, 100 JS-to-TS programs; budget: 16 times the source tokens

Pass: rustc compiles; tsc and ESLint pass. Naive checks only the complete program; self-refine retries with compiler feedback

Best-of-N: N independent attempts. S*: 8 candidates, 3 compiler-feedback rounds each



Up and to the left is better

Pass rate as the budget *k* grows

All 300 and 150 programs, self-refine: 72.3% → 82.0% (C to Rust), 33.3% → 46.0% (JS to TS)

DTV lessons for code translation

- Verify during decoding: fix an error before later code builds on it.
  - Adding DTV to Qwen3-4B with self-refine: compile pass 72.3% $\rightarrow$ 82.0% (C to Rust), with fewer generated tokens per case.

- **What the in-loop verifier checks is what improves.**
  - In C to Rust, DTV checks with rustc: compile pass rises, but the pass rate on differential tests against the C program shows no measurable change.

Code: `https://github.com/qsdrqs/dtv-translation`

Backup slides

Backup: why not patch after generation?

Coding agents generate the whole program, compile it, then patch the reported errors

	Post-hoc patch	DTV
First feedback	after the whole program	at each statement, block, or function end
Code after the error	written on top of the error	generated only from a passing prefix
Repair location	anywhere in the program	failing statement, widened to block, then function
Repair operator	patch	inline continuation, or a scoped patch
Model access	any model API	decoding control (open-weight models)

Follow-on diagnostics, DTV/naive (median): 0.50 on JS to TS (*n*=26), 1.00 on C to Rust (*n*=57)

Distinct diagnostics on shared error codes; naive's first attempt fails, one DTV attempt passes; Qwen3-4B

How do existing methods use a verifier?

Code comes with deterministic verifiers, such as compilers, type checkers, and tests: exact verdicts, with diagnostics that say where and why code fails

Approach	How it uses the verifier	What it misses
After generation best-of-N, self-refine, S*	checks complete programs; resamples, or retries with compiler feedback	an early error is found only at the end, after later code is built on it
Learned step scores process reward models	a trained model scores each step	a probabilistic score, with no repair hint; needs training data
Constrained decoding grammars, type automata	masks tokens that break the grammar or local types	cannot run the compiler or tests on unfinished code
In-loop backtracking ROCODE	rolls back on an analyzer error; makes the failed tokens less likely	the diagnostic text never reaches the model

When to verify: why not every token?

Verifiers need structurally complete code, not arbitrary token prefixes

```
fn main() {
    let count: i32 = "10".parse().unwrap();
    let total = count *
}
^ expected expression, found `}`
```

Decoding stopped after `*`, and a closing `}` is added so rustc can run. The code is unfinished, not wrong, yet rustc reports an error

rustc after each token of `let total = count * 2;`

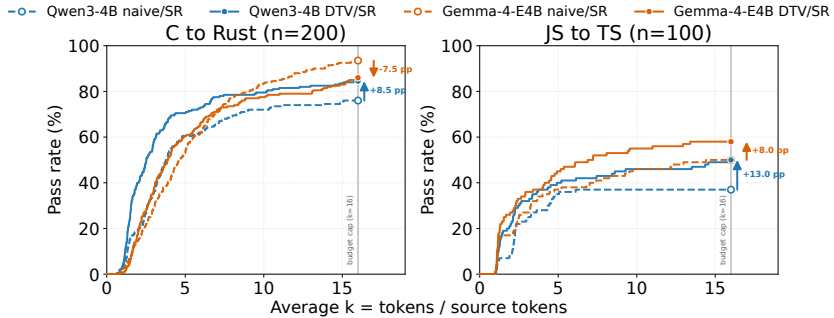
`let`✗ `total`✗ `=`✗ `count`✗ `*`✗ `2`✗ `;`✓

Six of seven checks fail on correct code, so DTV checks only where a statement, block, function, or program ends

Another model family: DTV leads early, not always at the cap

Qwen3-4B-Instruct and Gemma-4-E4B, self-refine; 200 C-to-Rust and 100 JS-to-TS programs

Cumulative pass rate as the per-case budget *k* grows; *k* = generated tokens / source tokens, capped at 16

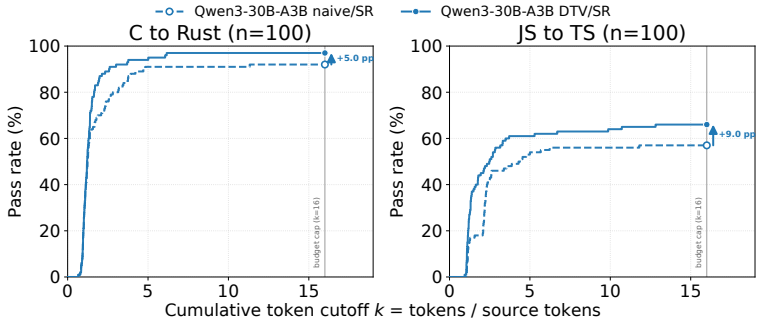


Gemma, C to Rust: naive ends at 93.5%, DTV at 86.0%; the curves cross near *k* = 7

A 30B model still gains from DTV, most on the first attempt

Qwen3-30B-A3B, a 30.5B-parameter mixture of experts; the first 100 programs per task

First attempt, no self-refine: 67% → **89%** (C to Rust), 18% → **48%** (JS to TS)



Curves: with self-refine

Removing any mechanism hurts, recovery most

Ablations on the first 100 C-to-Rust cases, Qwen3-4B; all four keep the self-refine outer loop and the same per-case token budget

- No-feedback: roll back without adding the diagnostics to the prompt
- No-escalation: always roll back to the failing statement
- Detect-and-abort: no in-loop recovery; the outer loop retries after the first failure

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

† not significant at n=100 (paired McNemar p=0.263); the other two drops are (p=0.035, p=0.003)

Gains concentrate on local fixes and save self-refine rounds

Where does DTV beat naive? Qwen3-4B-Instruct; 300 C-to-Rust and 150 JS-to-TS programs

Rescue rate by fix shape, one attempt each

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

Share of naive's failed first attempts that one DTV attempt solves (n). Fix shape: naive's own repair edited only the flagged lines (Local), some (Mixed), or none (Nonlocal). *: $n < 5$

Early intervention

On rescued cases, DTV emits fewer diagnostics on the same error codes: median ratio 0.50 in JS to TS, 1.00 in C to Rust

Self-refine rounds on cases both solve

