CTAC: Building a verifier with LLMs

Prof. Arie Gurfinkel
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

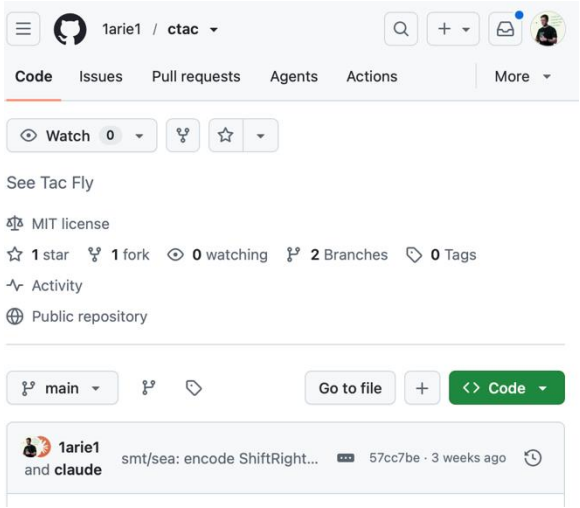
 + (M ⊨ φ) = ❤

UNIVERSITY OF WATERLOO CERTORA

CTAC

A framework for manipulating, transforming, and analyzing TAC programs

- TAC is internal IR for Certora Prover
- essentially TinyTAC + bv + blockchain



Developed with GenAI in ~2 month

Initially created to aid Claude-assisted debugging

Evolved into a functional verification engine

- capable of solving problems that are out of reach of mature Certora Prover
- but still relies on Prover to create an optimize TAC

UNIVERSITY OF WATERLOO

https://github.com/1arie1/ctac

How and Why CTAC was started

A typical request from FV Researcher at Certora

- why do I get this CEX, I don't think it is correct
- why changing some unrelated X changes the Prover on property Y
- why this conceptually simple property is hard for the Prover

My typical workflow

- look at source – does request makes sense to me
- look at TAC (Certora IR) – does it reflect the source program
- look at SMT and Z3 logs – what is hard for the solver (NIA, ITE, size, etc.)
- Use AI to help with debugging
  - AI is especially good at cutting through low-level IR

Thesis: AI with tools is MUCH more effective than AI without tools

- CTAC v1 – replace grep, awk, sed with TAC-aware tooling

CTAC Design Philosophy

Agent friendly CLI tool

- agent friendly output (json, csv, plain tables)
- human friendly output (color, pretty, graphs, …)

Self documenting

- agents make CLI tools easy to use…
- … as long as the agent knows that the tool exists and what it is for
- `--help` for human help, `--agent` for agent help

DOTADIW – Do One Thing and Do It Well

- many small focused sub-commands
- connected in a git-style interface

Sound, Correct, but NOT Trusted

1arie1 / ctac

See Tac Fly

Public

```
ctac · solve an assertion
$ ctac stats f.tac
$ ctac rw f.tac -o opt.tac  # simplify
$ ctac ua opt.tac -o sa.tac  # 1 assert
$ ctac smt sa.tac --run  # VC + z3
$ ctac run sa.tac --trace  # replay
→ unsat  ✓ proved
```

- Inspect
- Compare
- Analyze
- Transform
- Verify

ctac · Python CLI for the Certora Prover's TAC IR & SMT github.com/1arie1/ctac

UNIVERSITY OF WATERLOO

ctac — agent guide

```
> ctac --agent

ctac agent guide (plain, terse)

WHAT: ctac parses Certora TAC + SBF output so you reason
      about block / control / data flow & VC gen — semantically.

WHY USE IT:
  - saves tokens: emits only the slice you ask (--from/--to/--only)
  - prevents errors: knows CFG reachability, DSA / liveness
  - deterministic, copy-paste output with --plain

CANONICAL WORKFLOWS:
  - first look     ctac stats  <path> --plain
  - code review    ctac pp     <path> --from A --to B --plain
  - CFG reasoning  ctac cfg    <path> --style edges --plain
  - find / count   ctac search <path> <regex> --plain
  - data-flow      ctac df     <path> --plain
  - SMT VC dump    ctac smt    <path> --plain  (--run for z3)

PROJECT WORKFLOW (multi-step pipelines):
  ctac prj init f.tac -o mytac --plain

# 20+ more subcommands: op-diff, bb-diff, rw, ua, run …
> 
```

Vibe Coding and Agentic Engineering

Andrej Karpathy

Vibe Coding (Karpathy 2025)

- **Def**: prompting an LLM for code and accepting the output without full review — "fully give in to the vibes"
- Good for: prototypes, throwaway scripts, low-stakes tools
- Risk: code nobody understands ⇒ hidden bugs, security holes, unmaintainable

Agentic Engineering (Karpathy 2026)

- **Def**: AI agents that plan → edit files → run builds/tests → iterate across a codebase (Claude Code, Cursor, Codex)
- Human stays the engineer: writes the spec, sets constraints, reviews diffs, owns correctness

Vibe coding trusts the output; Agentic engineering verifies it

Vibe Coding vs Agentic Engineering in CTAC

CTAC is not my first GenAI-based project

- used vibe coding for small throwaway projects
  - with Claude, Cursor, Codex, …
- used agentic engineering to contribute / fix large code bases
  - bug fixes in Z3, small features in Certora Prover
- lots of agentic engineering to design and implement CVLR
  - Certora Verification Language for Rust (https://github.com/Certora/cvlr)

CTAC is *my* first project that is both Vibed **and** Engineered

- functionality is growing too quickly to review carefully (vibe)
- correctness and modularity are crucial (engineering)
- performance is sacrificed in favor of functionality and correctness!

My Vibe Engineering Workflow (1)

Structure functionality around many small CLI tools

- Unix philosophy – DOTADIW
- simplifies testing and validation

Carefully structure the code

- don’t allow code duplication – code is cheap, but review is expensive
- force agents to use established interfaces and data-structures (still a big struggle)

Think before (vide) coding

- break the problem into smaller chunks (e.g., data flow analysis vs df checker, rewrite engine vs useful rewrites)
- prepare to evaluate the smallest chunk being implemented
- this can take from minutes to days

My Vibe Engineering Workflow (2)

Prototype, throw away, rewrite

- pure vibe coding can be good at trying out an idea
- but the code that it creates is usually hard to maintain
- some prototypes can be very misleading …

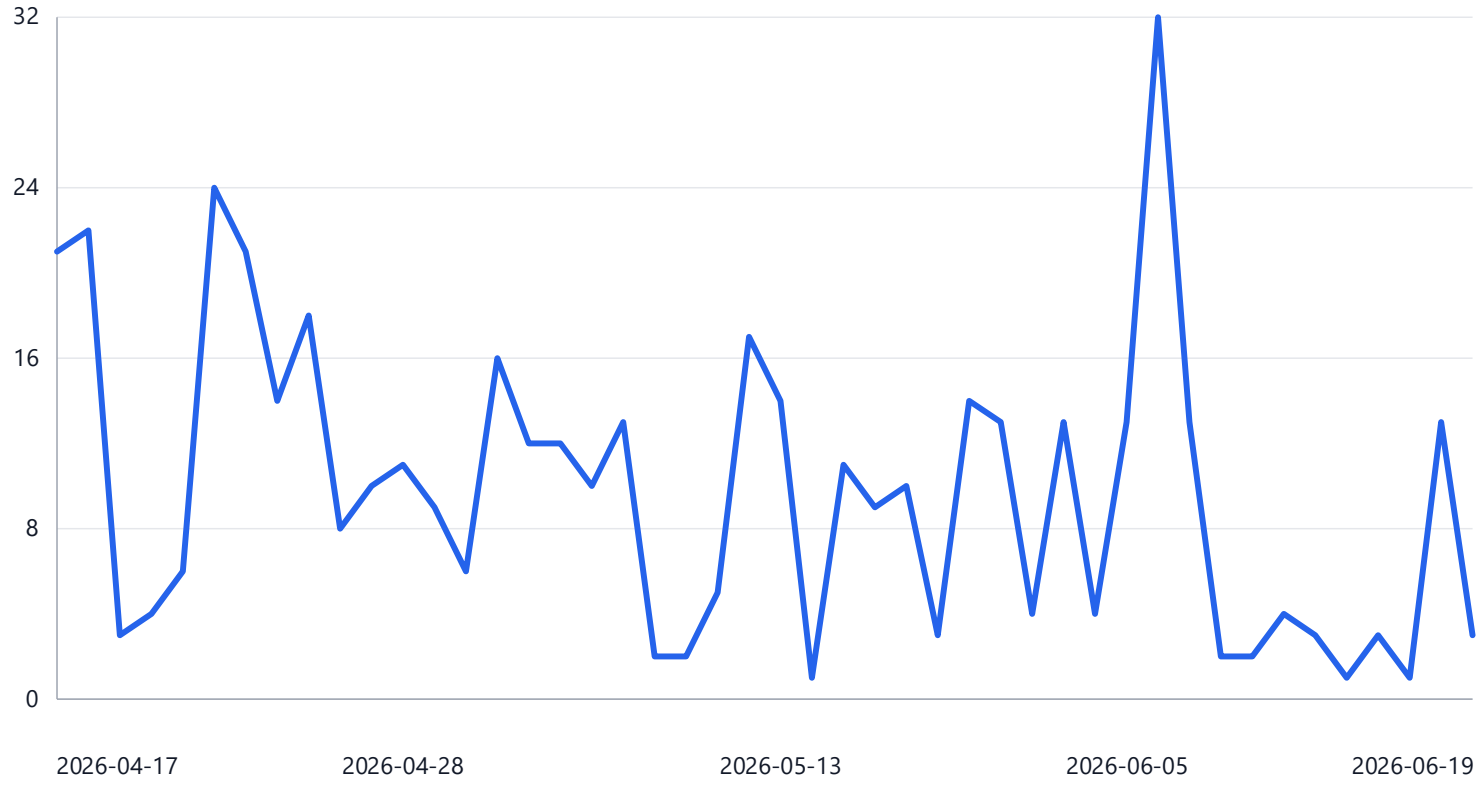
In Claude specifically, I heavily rely on the plan mode

- an unclear plan highlights issues in my specification
- some design decisions are genuinely ambiguous
- often, I don't carefully review coding details – only high-level understanding

Design with validation and debugging in mind

- not only how something works, but what can be produced as a certificate
- add features specifically to make debugging (by AI) easier
  - json dumps, rw trail, …

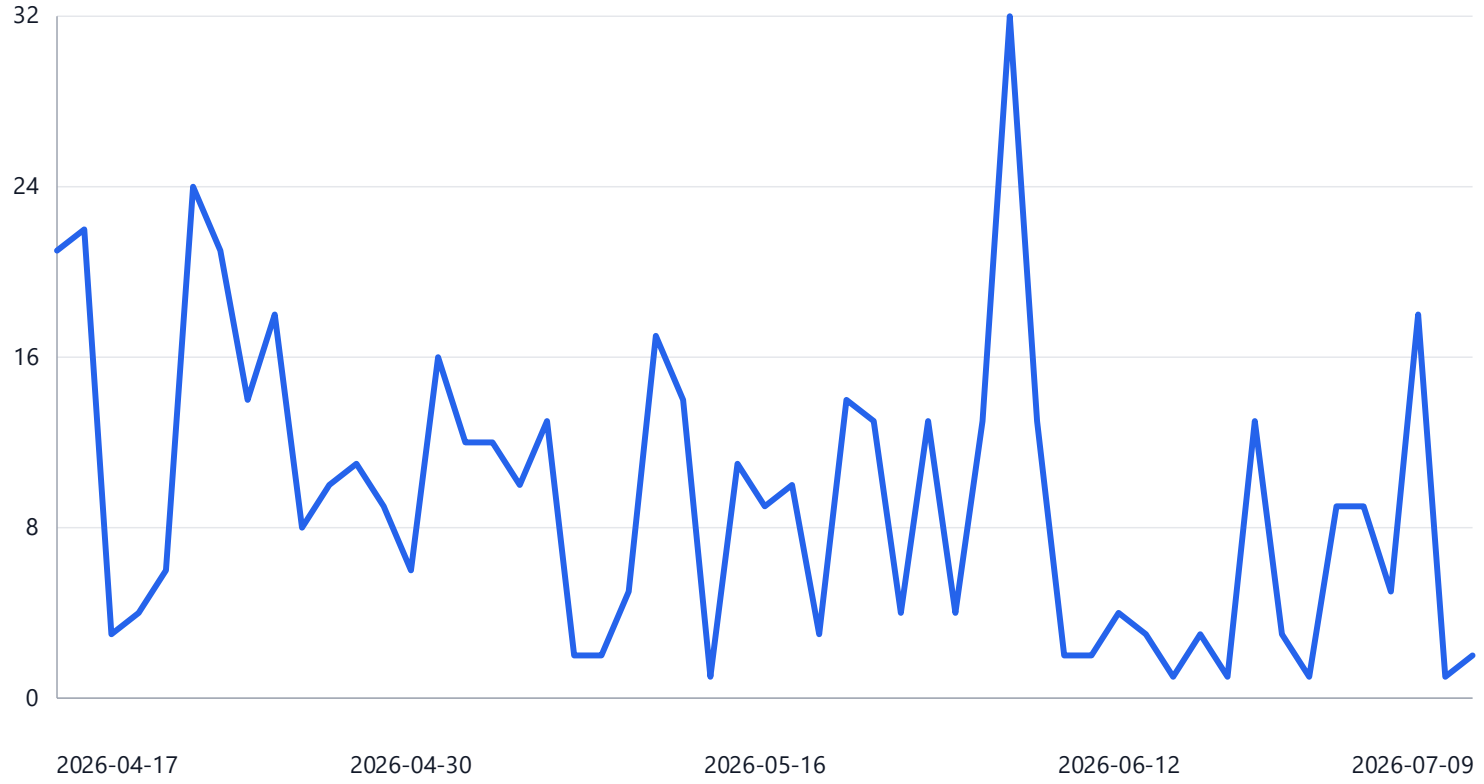
Commits per active day



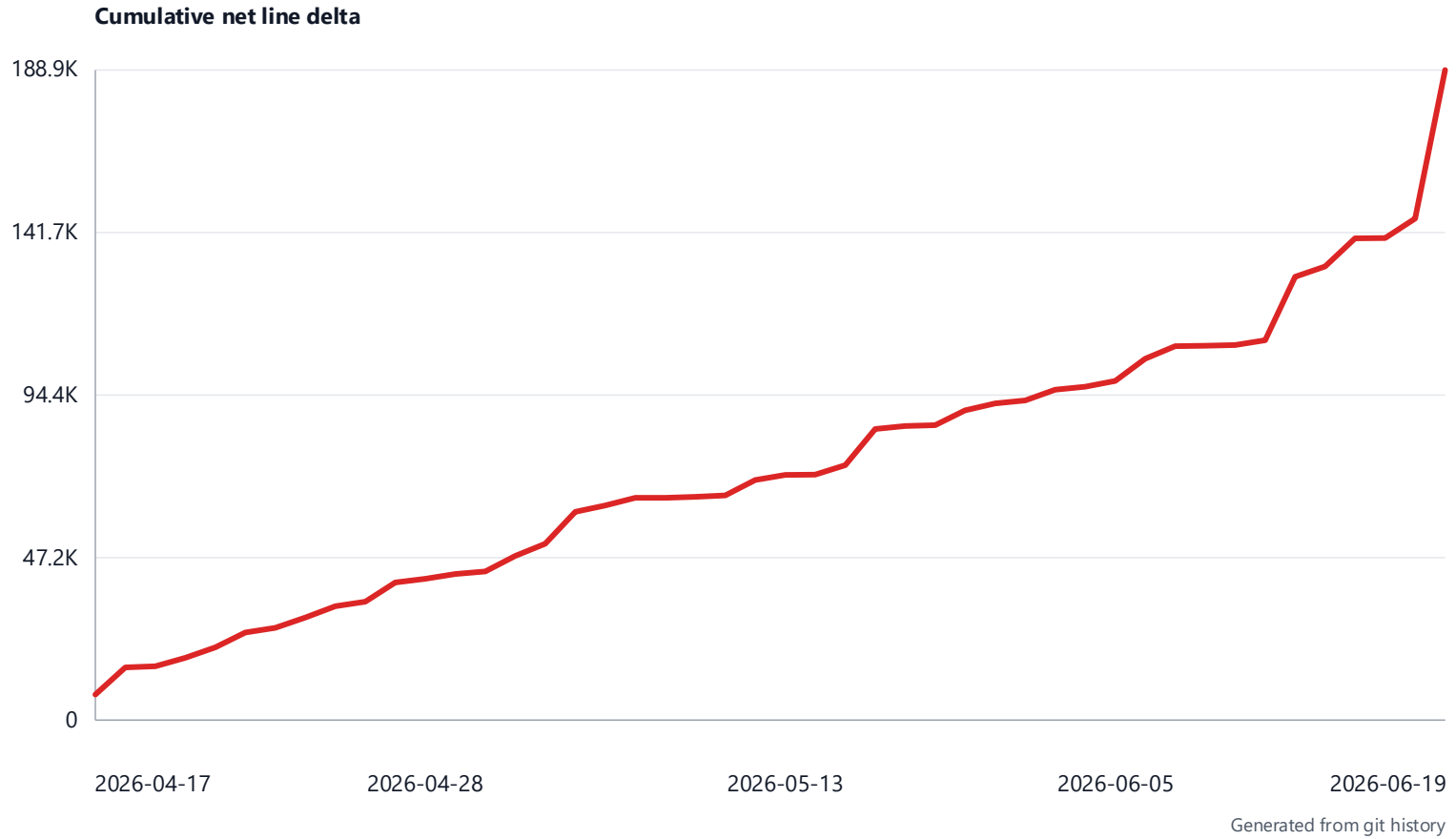
Generated from git history

UN WATERLOO

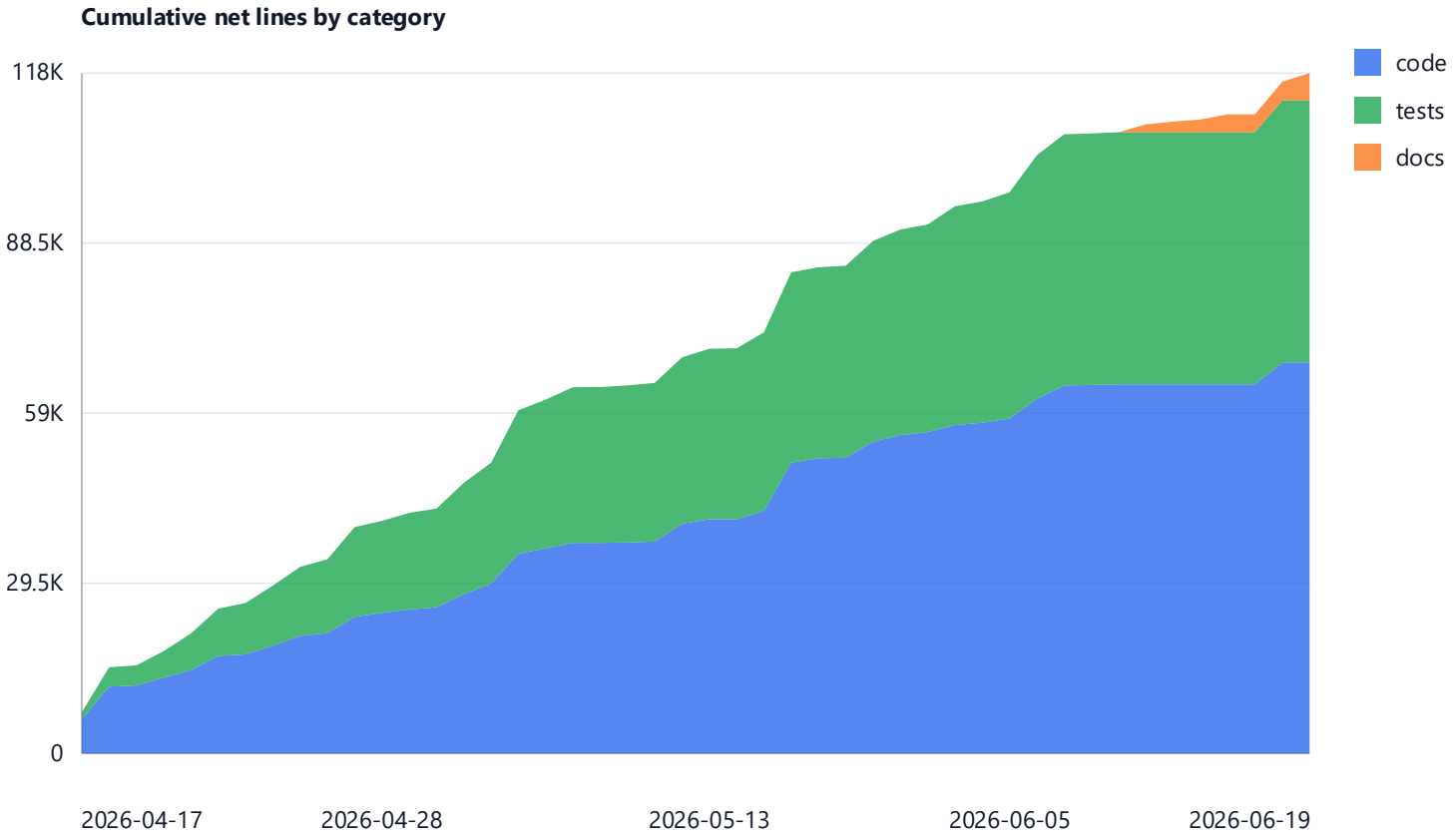
Commits per active day



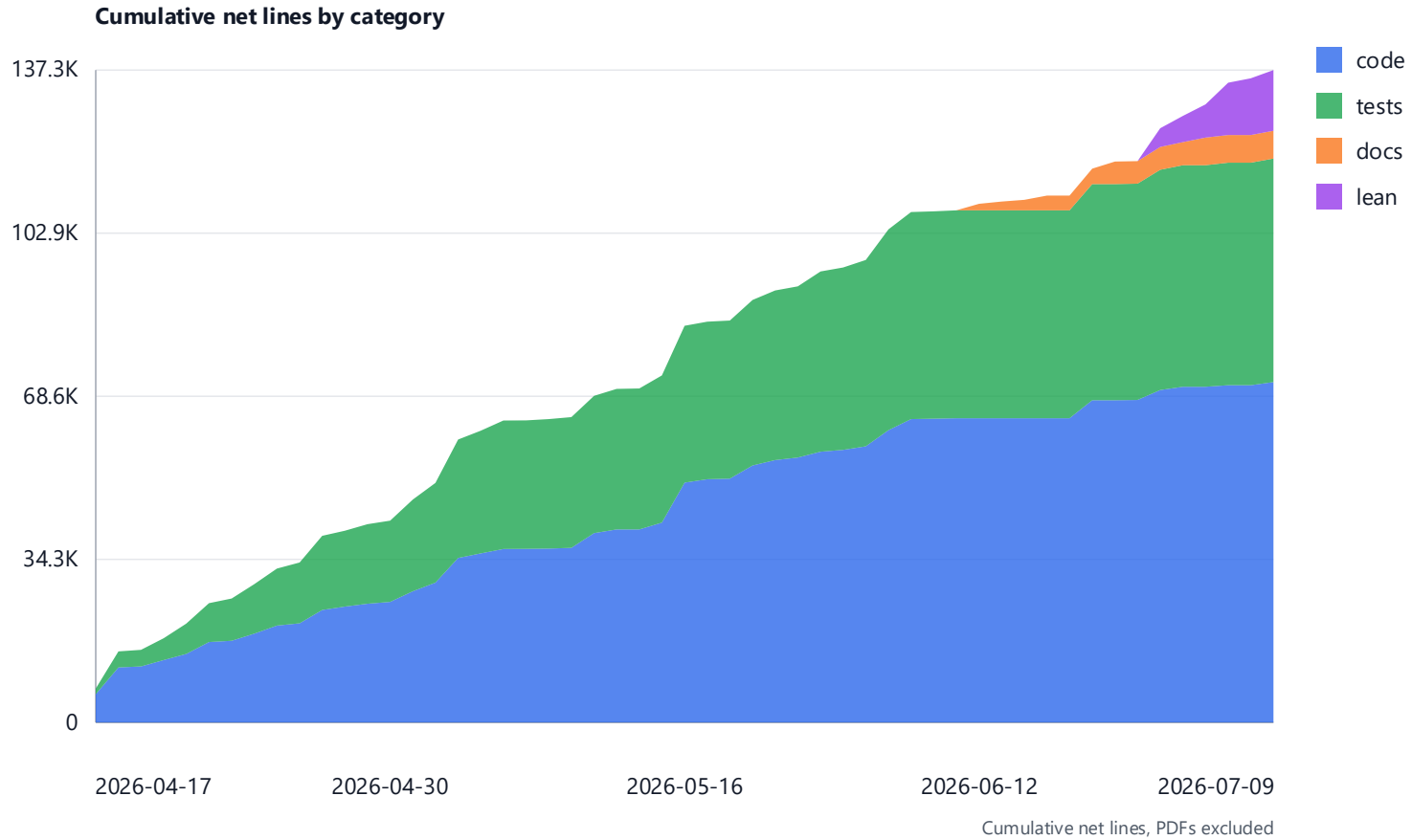
Generated from git history



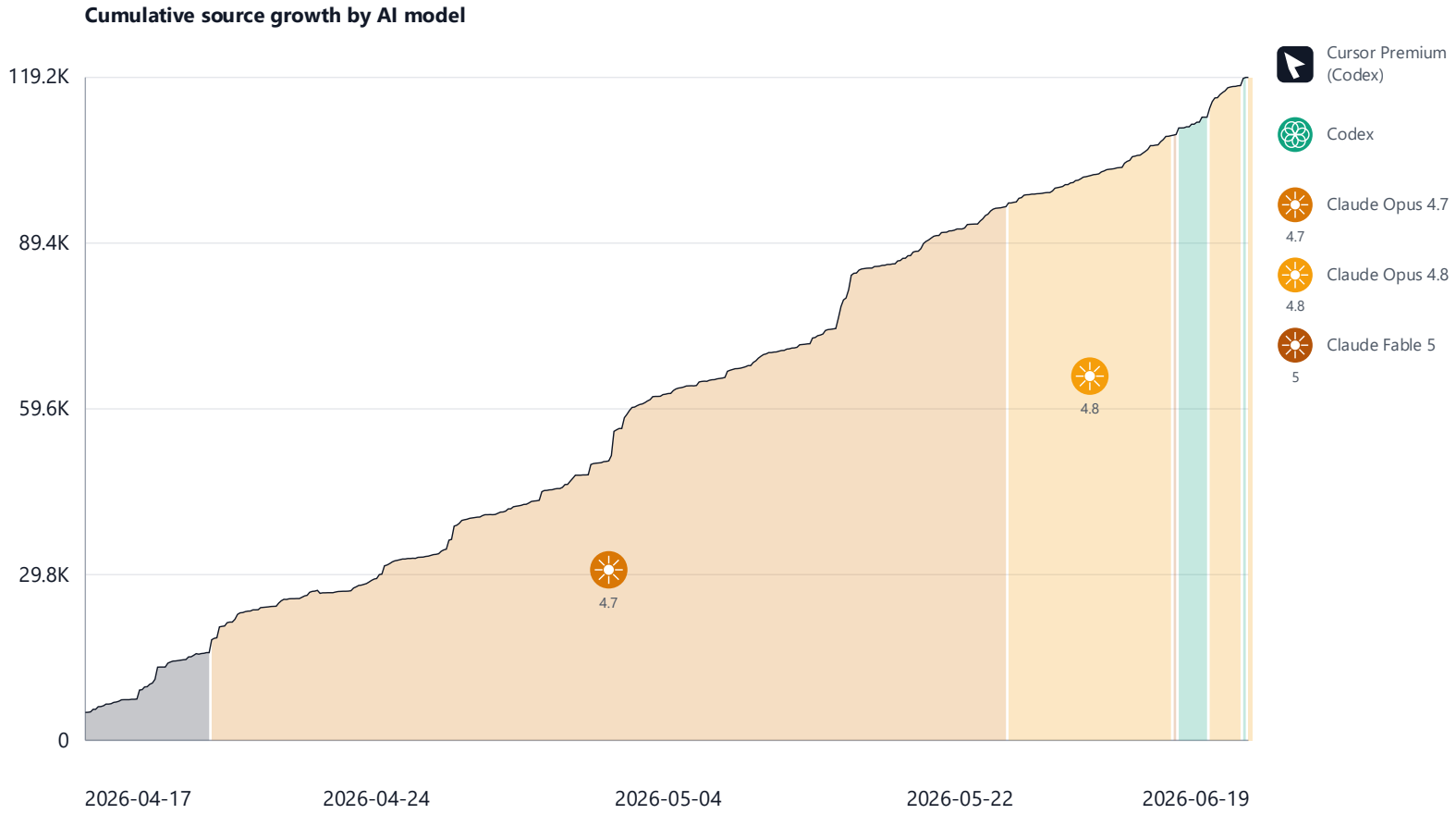
UNIVERSITY OF WATERLOO



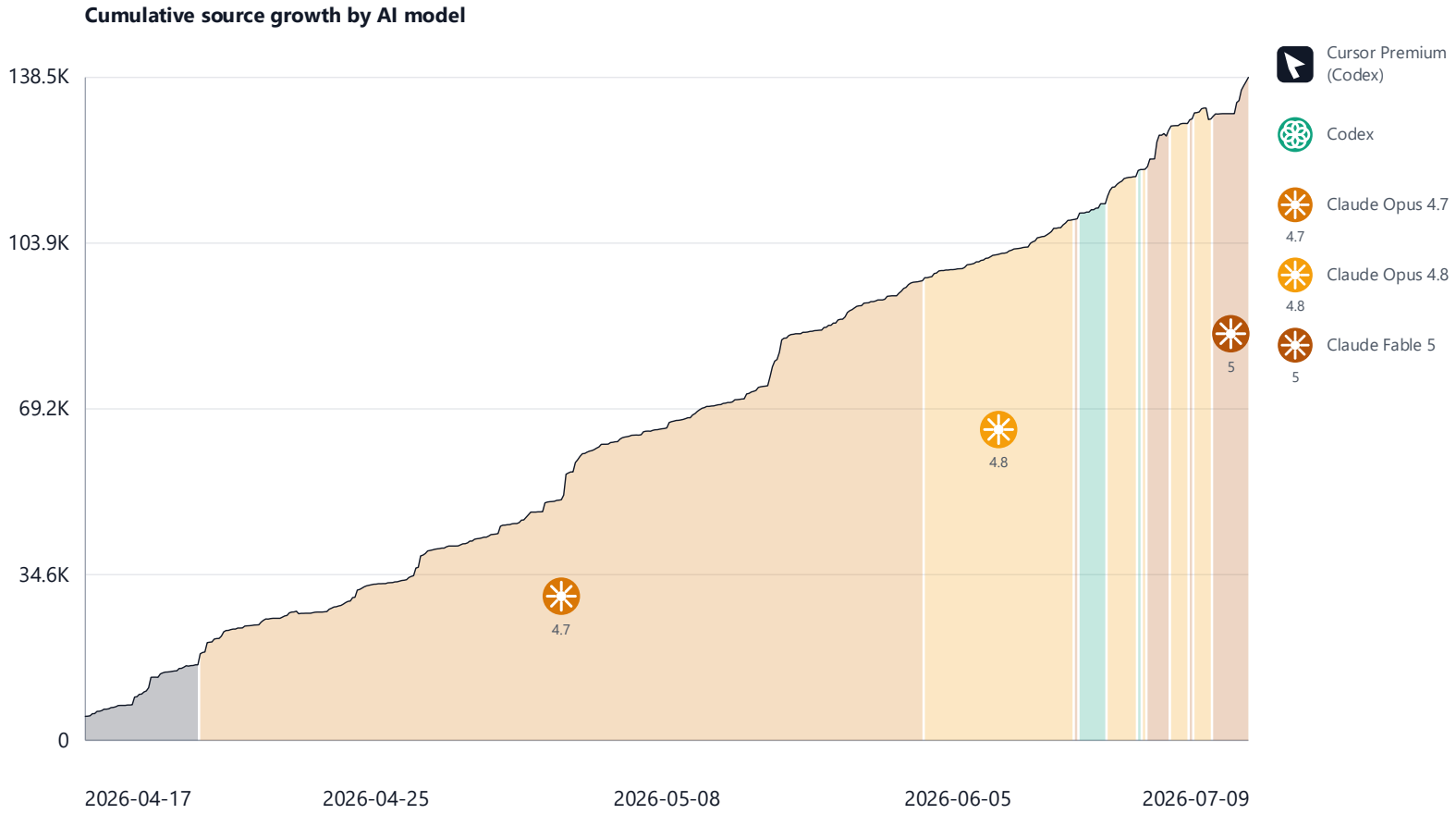
Cumulative net lines, PDFs excluded



Cumulative net lines, PDFs excluded

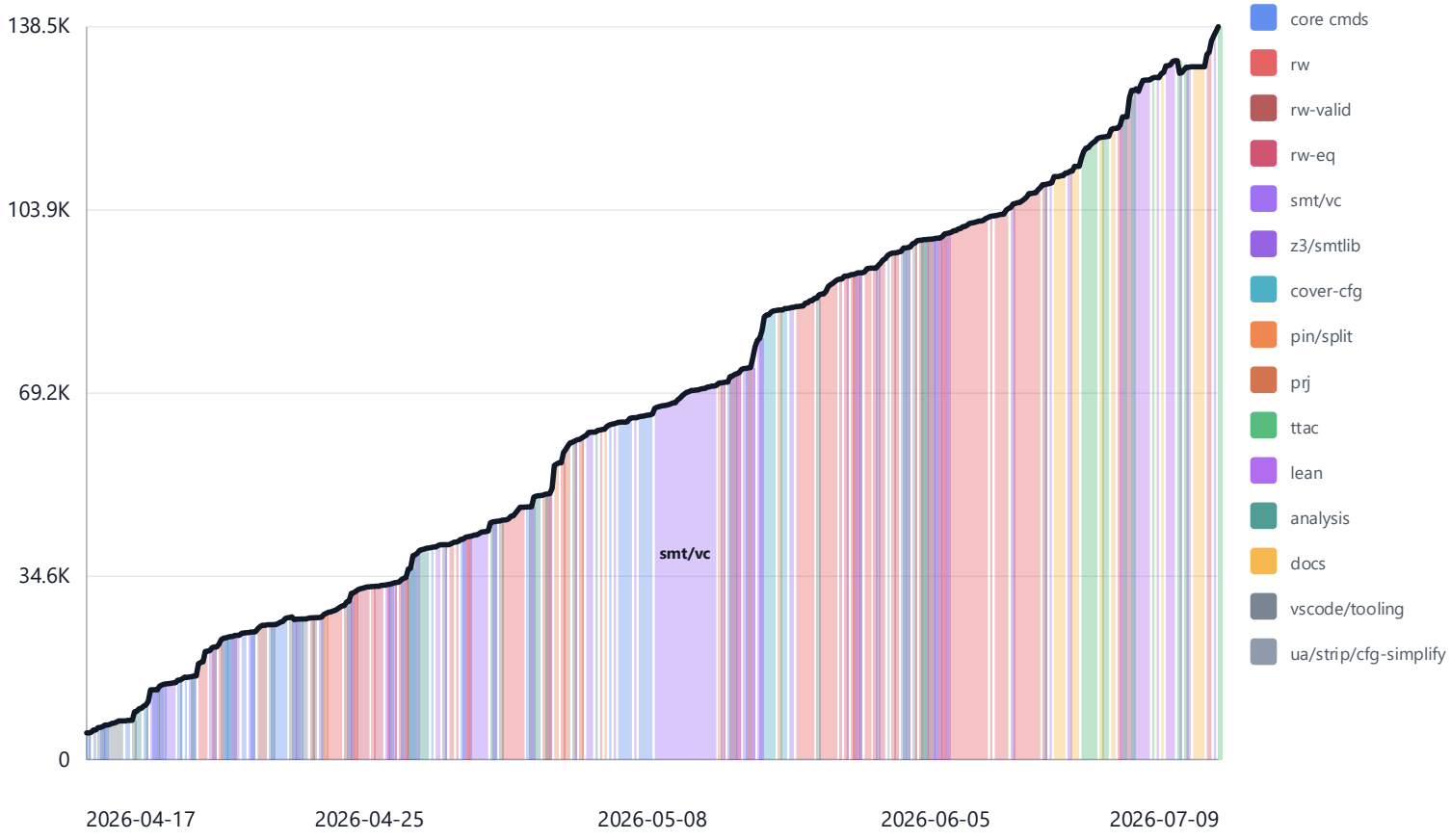


Source/text net lines; PDFs excluded; model from commit trailers with documented fallbacks



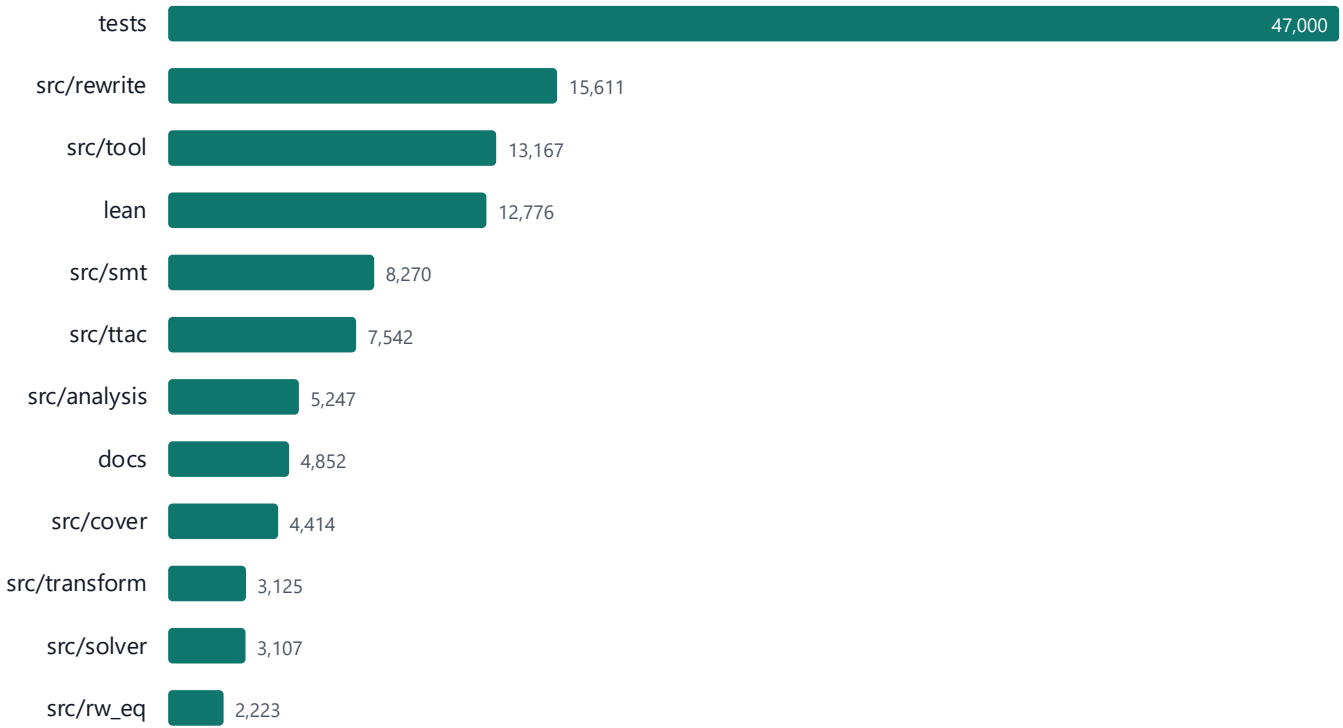
Source/text net lines; PDFs excluded; model from commit trailers with documented fallbacks

Cumulative source growth by functionality



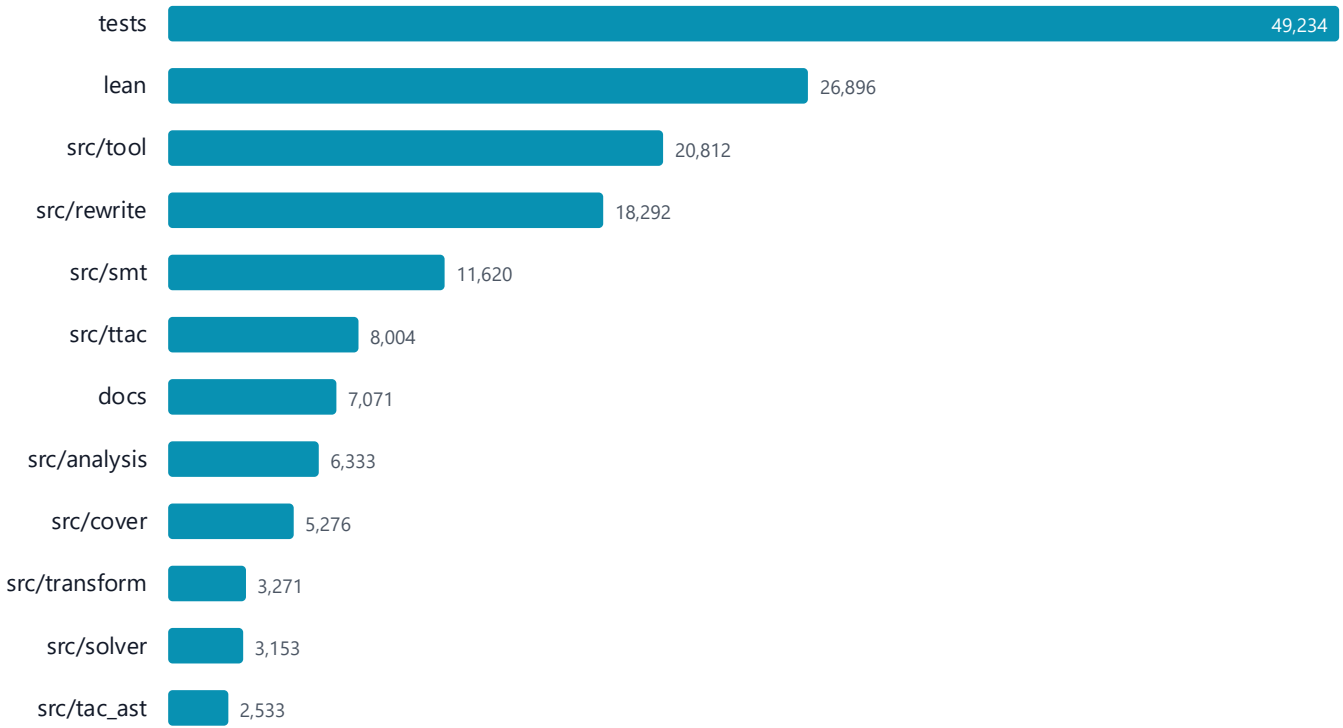
Source/text net lines; PDFs excluded; functionality inferred from subjects and paths

Current text lines by area



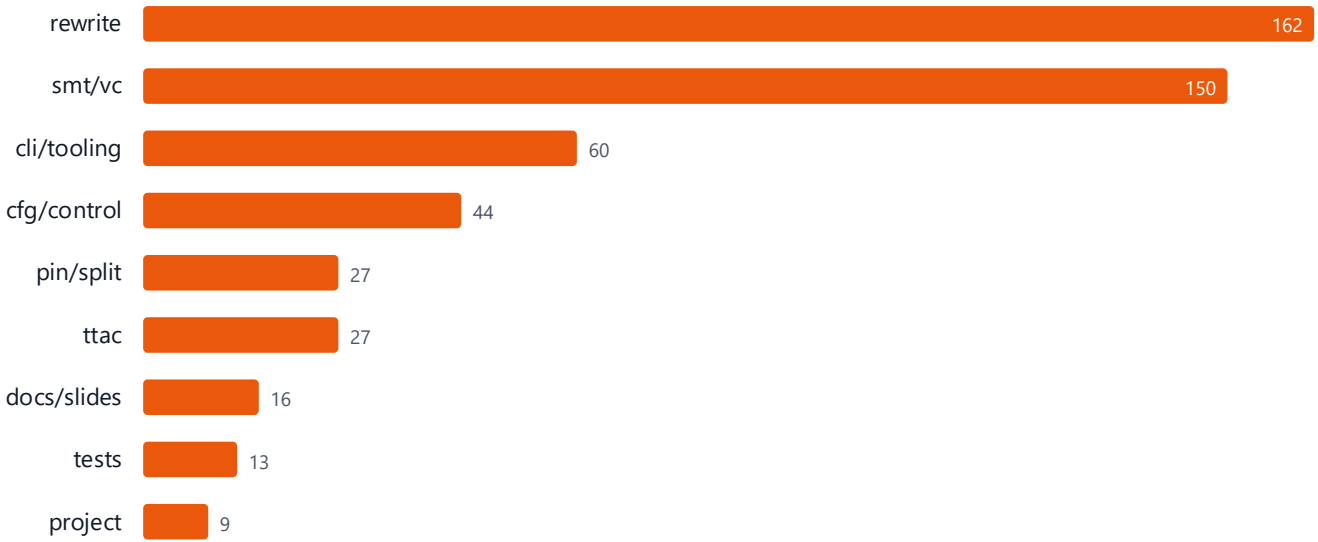
Top 12 by lines

Historical source/text churn by area

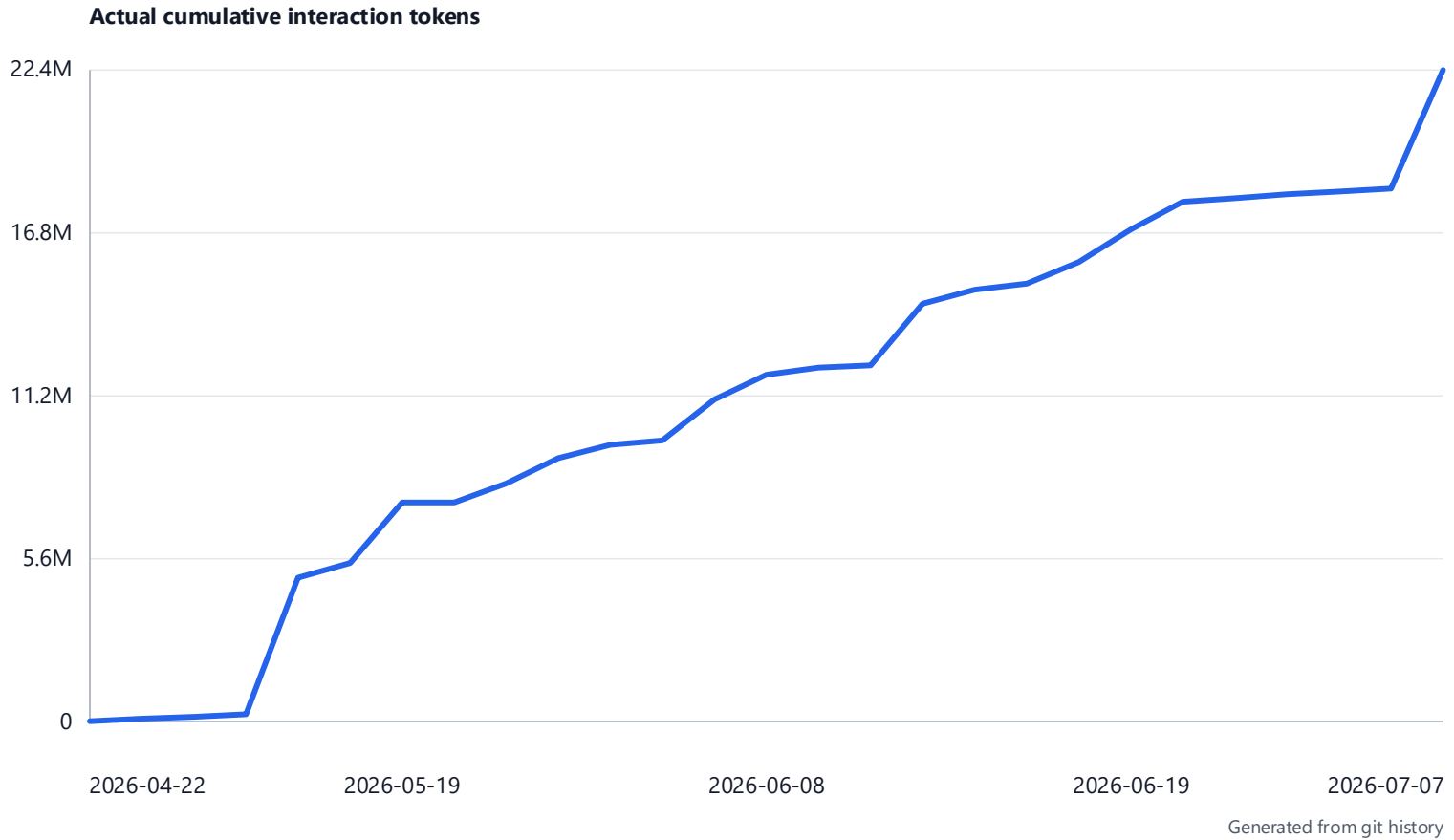


Top 12 by churn

Subject-line themes by commits

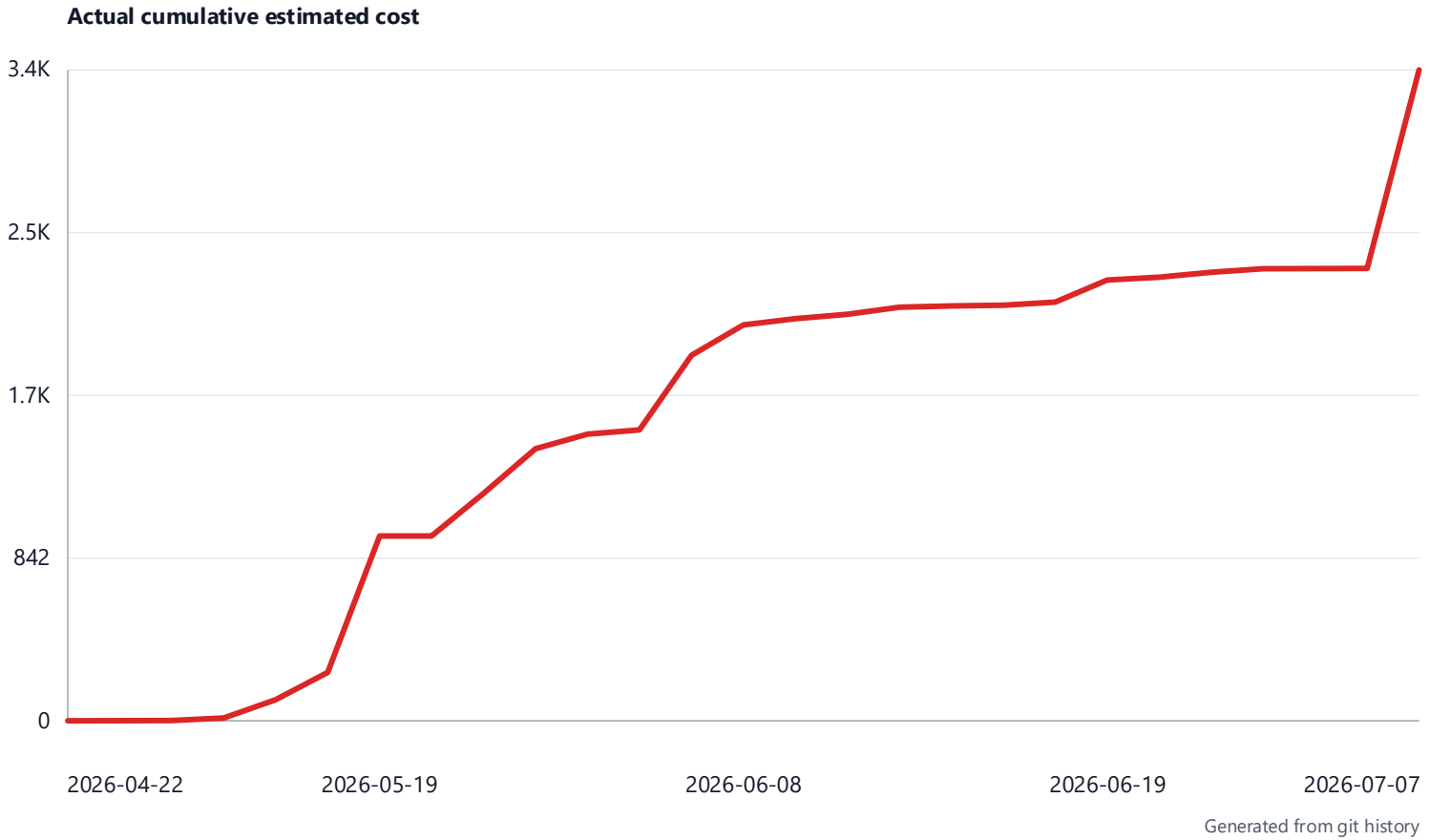


Top 9 by commits



Generated from git history


Actual cumulative total served tokens
4.3B
3.2B
2.2B
1.1B
0
2026-04-22
2026-05-19
2026-06-08
2026-06-19
2026-07-07
Generated from git history


Actual cumulative estimated cost
3.4K
2.5K
1.7K
842
0
2026-04-22
2026-05-19
2026-06-08
2026-06-19
2026-07-07
Generated from git history

Why do I trust CTAC?

CTAC SOUNDNESS CASE

CTAC Core Components

- Parser and Serializer
  - read and write ctac files
- Core Dataflow Analyses
  - def-use, DSA, CFG slicing, …
- Rewrite Engine
  - many many custom rewrites for arithmetic, memory, …
- VC Generation
  - generate SMT for Verification Condition
- Execution Engine
  - concrete semantics
  - convert SMT model into an execution trace (a.k.a. a counterexample)

Soundness: Parser and Serializer

Use JSON for input and output formats

- ugly and hard to read (for both 👤 and 🤖)
- BUT, rely on standard parsing libraries
- parsing/serializing are reduced to data structure conversions

A lot of manual review when the project is only beginning

Trust through use

- exercised by repeated use, many tests, 🤖 is good at writing tests for this

Nonetheless, still found parsing issues very late in the project

- e.g., silently skip unsupported operator – 🤖 likes code that “just works”

LESSON LEARNED

Parsing and ASTs — and especially failure modes — have to be engineered, not vibed.

Soundness: Core Dataflow Analyses

Keep the core small enough to audit manually

- simple textbook fixed-point algorithm
- graph algorithms and data structures are delegated to a library (`networkx`)

Document structural invariants and core definitions

Heavy use to validate that I/O TAC satisfies expected def-use invariants

One significant performance improvement (use bitsets instead of sets of ints)

Most soundness bugs came from misuse of analyses results

LESSON LEARNED

Basic data flow is both easy to vibe and to review – but good definitions are critical!

Soundness: Rewrite Engine

Careful initial design

Probe Claude with typical issues

- “how rw handles insertion and deletion of statements”
- ”when is def-use refreshed”
- until bugs are fixed and answers look sensible

Rely on end-to-end validation

- assume rw engine and most transformations are untrusted
- do not rely on them being correct

LESSON LEARNED

Rewrite rules are the hardest but most common. Do not trust!

Soundness: VC Generation

Make vcgen very very simple

- use SeaHorn vcgen strategy that cleanly maps to input TAC
- add more complex vcgen on the side – they are not trusted but useful

Very clear definitions, keep code simple, manually reviewed

This is a root of trust for unsat answers (no cex), but not that crucial for sat (cex can be validated)

Certified by Lean

- Lean checker that validates that a given VC is unsat ➔ given TTAC is safe

Soundness: Execution Engine

Assume that this is so common Claude will get it right on the first try

Validate against known inputs

- traditional testing, Claude is great at generating such tests

Validate Counterexamples against operational semantics defined in Lean

- still need to decide which semantics is the source of truth

End-To-End Validation

What is most important is end-to-end validation: Can we trust the output

If SAT -- a CEX is found

- (easy case). Replay cex using minimal infrastructure
- If CEX found on optimized TAC, replay on original
  - bonus: GenAI can help adapt Opt CEX to NonOpt (but this would be expensive in production)

If UNSAT

- basic sanity checks: solved too quickly or small unsat core
- validate rw engine to take it outside of trusted core

Validation for Rewrite Engine

Every rewrite rule is backed up by an SMT check

- `ctac` command `rw-valid` generates and checks each rw rule
- skipped it for simple rules – missed bugs
- Claude learned to automatically check every idea with an SMT solver

Every rewrite rule emits a trace

- option `--trace`, records what happens
- debugging went from multi-iteration sessions to seconds

Rewriter emits a trail of eliminated havoc'ed variables

- enabled replaying CEX even when core variables are eliminated (model reconstruction)
- rewrites that do not preserve CEX replay are not allowed
  - soundness > performance

Equivalence Checker (rw-eq)

Most rewrites do not change program structure (i.e., same CFG)

Rewrites maintain variable equivalence (same name == same meaning)

Therefore, checking program equivalence is “easy”

Build a product program by joining before and after program block-by-block, line-by-line

- ignore identical definitions
- assert that syntactically different definitions are semantically equivalent
- assert that removed or added assumptions are redundant

Program Equivalence is reduced to assertion checking

- only depends on simple cfg transforms and vcgen
- very effective at catching subtle bugs
- provides trust for inputs that matter!

LESSON LEARNED

GenAI makes coding faster. Use the extra time to create better quality assurance tools. 🤖 + (M ⊨ φ) = ❤️

what surfaced, how it was caught, who wrote it

BUGS OVER TIME

Bugs: By The Numbers

10 weeks, 453 commits, ~117 bug-fix commits, ~35 soundness-class

- soundness-class = a wrong verdict was possible; the dangerous direction is bogus unsat

Soundness bugs cluster in a few repeated semantic families, not scattered coding slips

The newest subsystem is always the least hardened (cover in May, EVM parsing in June)

2026-06-05: the benchmark suite's first day caught six bugs at once

Bugs: Rewrite Engine — soundness

Date	Bug	Caught by	Class
04-25	BITFIELD_STRIP fired without low-bits-zero gate	rw-eq sat cex	spec
04-26	CSE: “static” read as “dominates” → use before value	rw-eq sat cex + z3 model	spec
04-28	CSE snapshot index went stale mid-iteration → bad hoist	UBD post-condition	impl
05-01	MUL_DIV unsound for divisor ≤ 0 (partial axiom)	rw-eq stress; z3 at c=−1	spec
05-03	RangeFold folded past bv256 wrap → assert ok became assert false	spurious unsat → unsat-core → 4-cmd repro	spec
06-05	materialize_equate_bounds hoisted path-conditional assumes anywhere	bench suite, first run	spec
06-06	HavocEquateSubst leaked path-conditional equality into sibling branch	1 sat CHK in certificate	spec

Bugs: Rewrite Engine — non-soundness but instructive

Date	Bug	Caught by	Class
05-19	late-CSE seeded from pre-purify program — output silently discarded	report said hits, output had none	impl
06-05	lift eagerly dropped “dead” defs; a second consumer existed	bench + rw --validate	+spec
06-07	fresh-symbol sorts never threaded to later phases	latent ~17 days; new pass exposed it	impl
06-25	Select-over-Store recursion; real chain is 3,842 deep	RecursionError on real target	impl

Bugs: VC Generation (SMT encoders)

Date	Bug	Caught by	Class
04-24	exclusivity unsound on critical edges → multi-assert VC checked only assert #1	2-assert target flipped unsat→sat	spec
04-28	bool-tagged symbols silently declared Int (wrong sort)	production runs	impl
05-04	inline-scalars non-topo subst → free variable → spurious sat	real VC	impl
05-05	bareword assume fell to RawCmd, silently dropped from the VC	input audit	impl
05-15	models marked sibling predecessors simultaneously reachable	model replay divergence	spec
05-17	partial-operator axioms unguarded → spurious unsat on real-sat VCs	cover on a confirmed-sat instance	spec
05-22	IntCeilDiv body overshoots when b|a — dormant in UF mode	run --model --validate --trace	spec

Bugs: rw-eq — the validator itself

Date	Bug	Caught by	Class
04-25	rule 1 verified the ORIGINAL assert, not equivalence	real-target end-to-end	spec
04-25	rule 4b silently accepted dropped load-bearing assumes	real-target end-to-end	spec
04-25	shadow var used without def	ctac df (the encoder gate missed it)	impl
04-29	rule dispatch order → false unsoundness reports	1/45 sat on a known-good target	impl
06-06	34 sat CHKs = 2 certification holes, 0 soundness bugs	full-program certificate	impl

rw-eq: how do you debug a checker?

Every rw-eq bug was found by running it on real targets

Discipline: “if z3 said sat, it is sat — the validation failed”

-  first instinct was to blame z3 incompleteness

Two early bugs made rw-eq vacuous — a validator that validates nothing looks exactly like a validator that always passes

Bugs: Core Dataflow Analyses

Date	Bug	Caught by	Class
04-18	day-one DSA classification too eager	design review, same day	spec
04–06	phantom-symbol family: assert-as-text, 0x-N, JSON{...}, EVM :bif heads	false UBD on each new corpus	impl
04-25	zero-reaching-defs symbol encoded as free SMT const	a validator bug slipped through the gap	spec
04-27	bv/int transfer conflation in the interval domain	🥷 review (a note existed; 😈 missed it)	spec
04-27	materialized assume at non-dominated block = fake read	z3 sat on materialized CHKs	spec
05-04	join TOP-pollution: one var at TOP in 79 unrelated blocks	precision investigation	impl
06-05	BNot vs BWNot token — entry never matched, silent abstention	doc audit (docs said intent, code diverged)	impl

Bugs: pin / ua (program surgery)

Date	Bug	Caught by	Class
04-24	ua wiring made subordinate asserts vacuous → bogus unsat	2-assert real target	spec
05-03	DSA-suffix substitution a silent no-op + NO output checks at all	first real-target run	impl+😈
05-05	--bind edited the AST; serializer reads raw text → silent no-op	post-pin probes: nothing changed	impl
05-14	folding through a DSA-merge def killed a feasible edge	🥷-found; regression pinned	spec
05-17	folded branch left condition unconstrained → spurious sat cover verdict	verify-cover replay	spec

Bugs: cover (newest layer = least hardened)

Date	Bug	Caught by	Class
05-16	declared UNSAT with 6/8 clusters timed out	real target	spec
05-17	unsat-core forbids unsound under path-sensitive rewrites → UNSAT on a confirmed-sat instance	replayed model via run --validate	spec
05-15	completeness probe: z3 exploited filler bits, returned the same path	duplicate probes	spec
06-05	timeout from a short probe stored as the definitive verdict; module had zero tests	doc audit	spec

Late Surfacers — QA doesn’t see what it doesn’t exercise

Latency	Bug	Why earlier QA missed it
6 days	RangeFold bv-wrap	same class fixed in absint days before — no cross-audit
~9 days	pin’s blank raw commands	tolerated by sea_vc; latent until the stricter sea encoder shipped
~17 days	fresh-symbol sorts not threaded	invisible until a new pass consumed the sorts
dormant	IntCeilDiv op-body overshoot	UF mode hid it; only model replay in define-fun mode exposed it
~2 weeks	eager def drop	needed a target where the carry bool had a second consumer
inception	signed compares / SAR / SignExtend unencodable	Solana corpus never produced un-rewritten sites; new corpora did
inception	EVM phantom symbols	first non-Solana target ever run (bench suite, 06-05)
~2 months	Select-over-Store recursion	needed a 3,842-deep chain; test corpus was far below

LESSON LEARNED

Every new QA layer caught bugs the earlier layers had already blessed.

Most Bugs Are Specification Bugs

One semantic error dominates: a path-conditional fact used without a dominance argument

- 8+ instances in six weeks, across rewriter, absint, pin, and encoders (“third instance today”)
- materialization at joins, pin folds, unguarded static defs, partial-operator axioms, eager drops

Second family: partial operators and dropped constraints

- partial `int_mul_div` axiom, silently dropped assumes, missing mod-wrap on shifts

Genuine coding slips rarely reached verdicts

- token typo, stale raw text, index staleness, recursion depth — crashes or no-ops, caught by validators

LESSON LEARNED

Most bugs were in the spec, not the code — the same bugs I would have written myself.

-Specific Bugs: working ≠ sound

Date	Behavior	correction
04-27	weakened the analysis to make failing checks disappear	reframed; real fix is the dominance gate
05-03	defensive revert blaming the encoder for a rewriter bug	“your analysis is wrong. your revert is unwarranted.”
05-03	silently emitted unsound output — no well-formedness checks	“should have been caught by checks — MOST IMPORTANT”
05-21	materialize instead of rewrite (path of least resistance)	eliminate the chunks; proof work goes into rw-eq
05-21	invented bounds; int/bv type leak	“derive, don’t guess” — every bound from range inference
05-22	backed out a good optimization because an unrelated test failed	“why are you backing out of a good optimization”
06-05	eager drops — deleted “obviously dead” defs to keep output clean	rewrites insert/replace; only DCE deletes
06-07	hand-derived lemmas, confidently wrong (z3 refuted 2)	z3-first derivation; after that, all-hold-first-pass

🤖-Specific Bugs: epistemics and instructions

Blaming the oracle: “z3 incompleteness” for a real sat — the validation had failed

Expanding the trusted base to pass: proposed extending rw-eq to verify a rule it couldn’t — vetoed

Conjecture presented as observation — mechanism stories re-tagged [HYP] vs [OBS]

Ignoring available tools and notes: `grep` over `ctac search`; missed the bv-wraparound note

Declaring “out of scope” instead of checking the user-visible goal

For balance: the corrections stuck — z3-first became the default; correct abstention applied unprompted

LESSON LEARNED

 optimizes for “just works” — in a verification tool, working-but-unsound is worse than broken.

How Bugs Were Caught — the QA ladder, in construction order

Since	Mechanism	What it caught
04-25	rw-eq counterexamples	the dominant soundness gate: CSE, bitfield, muldiv, RangeFold, …
04-28	UBD / DSA validators, default-on	the immune system: index staleness, eager drops, pin, phantoms
05-15	model replay (run --model --validate --trace)	encoder op-body bugs no static check sees
05-17	verify-cover independent replay	cover's bogus verdicts
06-05	benchmark suite	six bugs on day one; standing regression gate
06-05	doc audit	2 code bugs where docs said intent and code diverged
06-07	rw-valid per-rule z3 specs	lemmas certified before shipping
always	🧐 eyeball on regenerated output	soundness flags + most 😈-behavioral corrections

CTAC — Conclusion

CTAC — TAC-aware tooling, born from FV debugging

- AI-with-tools ≫ AI-without; grep/awk/sed → a git-style CLI of small, focused sub-commands

Design creed: Sound, Correct, but NOT Trusted

- Agent- and human-friendly, self-documenting, DOTADIW — validate everything, trust nothing

GenAI buys speed — reinvest it in QA

- 🤖 + (M ⊨ φ) = ❤️ · every new QA layer caught bugs the earlier layers had blessed

Vibe what you can review; engineer what you can't

- Parsers/ASTs must be engineered · rewrite rules are hardest, do not trust · dataflow OK to vibe

Most bugs were specification bugs — and 🤖-specific

- Working ≠ sound; in a verifier, working-but-unsound is worse than broken