Purpose-sized languages: buying total verification with smallness

[author]
(preprint — authorship to be set before submission)

July 19, 2026

Abstract

Agents increasingly generate code, and the scarce resource is not generation but trust: who checks what a model wrote, and whether a third party can re-check it. We argue that verification, not the harness around a generator, is the durable layer of the agent stack, and that its leverage grows with model capability rather than falling with it. We develop a concrete form of the thesis: the smaller a language's sanctioned surface, the more of a program's behavior a checker can decide — a direction we demonstrate at two purpose-sized points, not a scaling law measured across a range of sizes. A general-purpose compiler checks types but cannot say whether a program halts, what it may touch, or how much it may emit; a language sized to a purpose can make those properties mechanical and complete.

We present `litelite`, a kit of eight zero-dependency crates (the workspace totals 8,214 LOC across eleven crates with the three languages built on it; caps CI-enforced, native and `wasm32-unknown-unknown` both green) that pays the shared kernel of a family of purpose-sized languages once — a parser depth guard, a UTF-8-safe lexer, one fuel type, one capability table — each an invariant the three parent languages implemented divergently or omitted. On the kit we build `prooflite`, a total fuel-bounded reference language, and `stratlite` with its verifier `backtestlite`, a strategy language whose backtest reduces to one reproducible integer and which has no name for a future bar. The demonstrated saving is the base kernel: it is reused across the two kit languages rather than re-rolled per language, so each lands at its language-specific LOC only (1,955 and 2,562 LOC) in single recorded git sessions, each surviving adversarial review. We flag the contrast against the parents' 3,100–8,000 LOC as suggestive rather than clean: N = 2, the same developer and model era built both kit languages and all three parents, and the kit languages are tree-walk with no codegen while the compiling parents carry emitters — so only the kernel amortization is on firm ground (§4.5).

We put the verifier to work as a selection filter over 134 agent-generated programs (100% compile, 98.5% survivor) and specify it as a training reward with tested anti-hacking guards. We report the negative results plainly: on our single month of one asset the fuel bound never bound (the most expensive strategy used 0.74% of budget) and the held-out window is no harder than train (a 1.5-point gate-clear gap), so the benchmark has no out-of-sample teeth; and the verifier certifies well-formedness but not profit — 114 of 132 survivors lose money on the window they were scored against. We then run the verifier as the sole training reward: a small open-weights model (`Qwen3-0.6B`), fine-tuned by verifier-only rejection-sampling self-play with no teacher model and no API key, goes from a measured-zero floor — zero compiling programs in 256 attempts, because `stratlite`

is in no pretraining corpus — to 100% compile and ~96% held-out gate-clear, holding across a five-month embargo and a never-trained asset (95.7% / 96.5% / 96.1%). This carries the predecessor tempo-x402 recipe off `rustc` onto a purpose-built language — though it measures generation validity, not tempo-x402's task-solving pass@1, and the measured-zero floor reflects the language's absence from pretraining rather than a harder task; the near-zero train-vs-held-out gap marks the lift honestly as grammar competence, not out-of-sample edge, which stays the selector's job. Run a second time on `prooflite` — a data-free compute language, via the same trainer and ladder shape with a language-specific reward binary (different success rung and styles) — the lift repeats, from 3.5% to ~95% rich generation with ~100% of it novel against the cold-start corpus, so the result is not a one-grammar artifact (generality past the shared base model and kit stays untested). A held-out problem-solving benchmark (30 specs, exact-output match) then closes the pass@1 gap and returns a two-sided verdict: transfer is real — plain SFT on the 174 verifier-selected corpus programs lifts solving from 20% to 80% pass@8 — but the self-play that wins generation (94.5% vs 48.8% RICH) gives half of that back (43.3%), and the loss localizes exactly to problems whose correct output conflicts with the reward's shape, with the answer computed and then padded: the policy internalizes the verifier's preferences as unconditional habits that override the spec. A fourth arm confirms the mechanism by fixing it: swap the reward's top rung from output shape to output CORRECTNESS on 48 training specs (families disjoint from the held-out hard tier) and the same recipe reaches 93.3% pass@8 — above plain SFT, including hard families it was never trained on — while its unconditional generation drops to 24.2% RICH. Each arm maximizes exactly its reward and degrades the other axis: self-play makes the model more like its reward, everywhere; a verifier certifies what you ask it to certify, and the policy becomes it. A third language applies the lesson constructively: with a BEHAVIORAL reward (event scripts over a UI-app language), one hour of the same recipe takes the same model from an exactly-zero floor to 16/16 held-out app specs at pass@8 — the loop shipping as a working local app-builder, not a benchmark. One experiment remains unrun — a frozen-model A/B arm for which there is no API key — and ships as an instrument with pre-registered commands, marked PENDING. The deterministic verifier plus committed artifacts reproduce every number in this paper from a command — including the fine-tune's scoring, from committed sample pools and candles; the generation itself (agents, a frozen API model, or the fine-tune) does not, and is committed or recorded rather than regenerated.

---

1. Motivation

Agents increasingly write programs, and the durable question is not how well a model writes but who checks what it wrote. Most of an agent harness is machinery that encodes an assumption about what today's models cannot do: a retry policy, a scaffolding step, a prompt convention. Those assumptions go stale as models improve, so the harness layer is perpetually rewritten and rarely compounds. Verification is different in kind. Its job is not to compensate for a weak generator but to establish trust across an adversarial boundary, and that job does not get easier or harder as the generator changes — it gets more valuable as more code arrives from parties you cannot audit by hand.

The distinction we build on is testimony versus physics. A model reporting that it checked its own work is testimony: it is only as trustworthy as the model, and it degrades exactly when you most need it, under adversarial pressure or distribution shift. A compiler rejecting a program is physics: the rejection holds regardless of who or what produced the program, and it is cheap, deterministic,

and independently re-checkable by anyone who distrusts the checker. As agents become economic actors that exchange code and commit to its behavior, third-party-checkable verification becomes the layer that carries trust, and its worth rises with model capability rather than falling with it. Harnesses melt; verifiers compound.

The empirical seed for this thesis comes from the predecessor project, tempo-x402: fine-tuning a 0.5B model on compiler-verified self-play lifted pass@1 from 1.5% to 16.4% on a 201-problem benchmark, with the Rust compiler serving as a free self-play verifier (prior work, tempo-x402; not reproduced in this repo). That result used one verifier on one task family, and it hinted at a more general claim we develop here: the leverage lives on the verifier side, and it grows as the checked properties get stronger. A general-purpose compiler checks types and borrows but says nothing about whether a program halts, what it can touch, or how much it can emit. This paper asks what a verifier can guarantee when the language is sized to the purpose rather than to generality, and treats that verifier — not the language — as the product.

---

2. The kit

The kit is the shared kernel of the lite family — the pieces that `rustlite`, `soliditylite`, and `bashlite` (~19K LOC of purpose-sized languages inside the parent project, `localharness`) each hand-rolled separately, extracted and paid for once. It ships as eight crates unified by the `litelite` facade: four foundational kernel crates, the capability layer `caplite`, and two independent emitters. The full workspace is eleven crates — the three languages of §3–§5 ride on top — and every one of them declares zero external dependencies: the only entries in any crate's `[dependencies]` are other crates in this workspace (verifiable by inspecting each `crates/*/Cargo.toml`; `std` only, native and `wasm32-unknown-unknown` both green). The whole workspace is 8,214 lines against a 25,000-line ceiling (`bash scripts/caps.sh`).

crate	role	LOC	tests
`diaglite`	spans, coded diagnostics, caret snippets	252	7
`lexlite`	UTF-8-safe byte-cursor lexer kit	290	7
`parselite`	recursive-descent harness + depth guard	238	4
`fuellite`	fuel + byte budgets	180	4
`caplite`	host capabilities as data	521	10
`evmlite`	EVM assembler + diff-oracle interpreter	1,400	17
`modlite`	wasm module builder	770	8 + 1 doctest
`litelite`	facade re-exporting the seven above	—	1

LOC from `bash scripts/caps.sh`; test counts from `cargo test -p <crate>`. Fifty-eight unit tests plus one doctest cover the kit.

2.1 Invariants paid once

Each foundational crate is a single invariant the three parents implemented independently, with divergent bugs to show for it. The kit's value is not that these pieces are clever — they are deliberately boring — but that each bug class is now closed in one place.

- **The depth guard.** `parselite` defines `DEFAULT_MAX_DEPTH = 96` once

(`crates/parselite/src/lib.rs`). `rustlite` and `soliditylite` each carried a near-verbatim ~120-line copy of the parser harness, `MAX_RECURSION_DEPTH = 96` duplicated in both. `bashlite` had **no guard at all** — nested `if`-in-`if` recursed one frame per token until the stack overflowed, which on a wasm stack is an uncatchable abort that kills the browser tab rather than returning a diagnostic. On the kit a language cannot forget the guard, because `enter()` is the only way into the cursor.

- **The mojibake bug.** `lexlite`'s `next_char` decodes UTF-8 rather than

byte-casting. The same multi-byte-character bug was fixed twice in the parents, differently each time; the kit's cursor is UTF-8-safe by construction, and it makes the nested-versus-flat block-comment choice and the digit-underscore-separator flag explicit rather than silently divergent.

- **Three unrelated fuels.** `fuellite` is one `Fuel` type

(`burn` / `Exhausted`) plus one `ByteBudget`. The parents had three unrelated implementations of the same idea: `bashlite`'s per-statement fuel with 256 KiB output caps, `soliditylite`'s interpreter `STEP_BUDGET` with a memory cap, and `rustlite`'s parse-depth cap backstopped by an out-of-band JS watchdog (`crates/fuellite/src/lib.rs`). The composition rule the parents learned the hard way is now structural: one `&mut Fuel` threads down into every sub-evaluation, so fractal composition still terminates.

- **The triple-declared host table.** `caplite` makes one `CapTable`, declared

as data, drive four consumers that the parents kept in sync by hand: typed signatures for the checker, import emission for codegen, human docs, and a versioned parity manifest with an FNV-1a-64 hash for the far side of a boundary. `rustlite` hand-synced a Rust capability table against a JavaScript worker's copy, and the two drifted repeatedly. `caplite` kills that drift as a class — one declaration, or a hash mismatch.

2.2 The constitution as mechanism

The parents' recurring lesson was that a behavioural rule only holds once it stops being a convention and becomes a gate. The kit encodes three.

- **Caps are CI, not etiquette.** `scripts/caps.sh` enforces $\leq$2,000 LOC per

crate, ≤25,000 repo-wide, and ≤8,000 characters on `CLAUDE.md`, and CI runs it. At a cap the rule is split, shrink, or kill — never raise the number. The cap binds in practice: `prooflite` sits at 1,955 of its 2,000 lines and `stratlite` at 1,893, and the same 2,000-line cap is what forced the M4 split into the `stratlite` language (1,893) and its `backtestlite` verifier (669) rather than one crate straddling both. The tightest cap is the surface one: `CLAUDE.md` is 7,963 of 8,000 characters (all four figures from `bash scripts/caps.sh`). The deeper invariant the character cap protects is that the repo's honest map stays inside one context window.

- **The depth guard is the only door.** A language cannot enter the parser

harness except through `enter()`, and `guarded()` pairs `enter`/`leave` on every path, so the recursion bound cannot be bypassed by forgetting to restore depth on an error return.

- **Fuel is the only loop.** An evaluator that burns fuel on every step is

mechanically total — "this program halts within N units" holds by construction, not by review. There is no unbounded loop primitive to reach for; the budget is the loop.

2.3 Why the emitters stay separate

Constitution rule 4 forbids a unified codegen trait, and the two emitters honour it by being independent crates with no dependency between them (neither `crates/evmlite/Cargo.toml` nor `crates/modlite/Cargo.toml` names the other). This is not tidiness — it is a claim proven in the parents that absolute-jump machines and structured control flow are semantically irreconcilable. `evmlite` is an EVM assembler over an opcode SSOT with minimal-width `PUSH` and two-pass `PUSH2` label back-patching — labels resolve to absolute byte offsets — paired with a diff-oracle interpreter that performs real `JUMPDEST` analysis and rolls back storage on revert. `modlite` builds wasm modules with LEB128 encoding, functype interning, locals run-length encoding, and the index bookkeeping wasm's relative, structured control flow imposes (imports occupy the low function indices, so `modlite` makes importing after a function exists a hard error rather than a silent index shift). A shared emitter trait would have to abstract over "jump to this byte" and "break out of this block," and forcing a common shape onto them is precisely where the parents miscompiled. They ship as separate libraries, and a wasm backend that never touches EVM depends on only one of them. Both emitters are built and tested but as yet **unconsumed within this repo**: neither kit language emits code, so their named consumer is the pending M5 parent re-homing (§6.4). Their payoff is argued from the parents, not demonstrated here.

---

3. prooflite: the reference total language

`prooflite` is the kit's first language and its existence proof: the smallest program that drives every kernel crate end to end — `lexlite` lexing, a `parselite` depth-guarded parse, a `fuellite`-fueled tree-walk evaluator, and a `caplite` capability table as the host seam — with every failure a coded, spanned `diaglite` diagnostic. It is deliberately not in the `litelite` facade. It is a consumer of the kit, the measured answer to "what does a language on the kit cost, and what does it buy?"

The whole language is 1,955 lines of Rust including its tests, against the 2,000-line constitutional cap (`bash scripts/caps.sh`), with 31 unit tests and 2 doctests all green (`cargo test -p prooflite`). Nothing here is large. That is the point: the guarantees below are available *because* the language is small enough to reason about completely, not in spite of it.

3.1 The language

Values are 64-bit signed integers and booleans — nothing else. Statements are `let x = e;`, `x = e;`, `print e;`, `if e { ... } else { ... }` (else-if chains of any length), and `repeat e { ... }`, whose count is evaluated once, up front. Expressions are literals, variables, host-capability calls `name(a, b)`, the unary `-` and `!`, the usual arithmetic, comparison, and short-circuiting logical operators, and parentheses. Comments are `// line` and nested `/* block */`. There are no functions, no recursion, and no `while`: the only loop is `repeat`, and it must announce how many times it will run before it runs.

Three guarantees fall out of this shape, and each is a property that a general-purpose language cannot give mechanically:

- **Termination.** Every statement, every expression node, and every `repeat`

iteration burns one unit of fuel from a single tank; a host call burns one plus its declared cost. When the tank is dry the program stops with `E0206`. "This program halts within `limits.fuel` steps" therefore holds for every program, adversarial ones included, by construction rather than by inspection. The default budget is 100,000 fuel and 64 KiB of output (`crates/prooflite/src/lib.rs`, `Limits::default`).

- **A complete effect bound.** The host's capability table is the entire world

a program can reach. Calls resolve, type-check, and cost fuel against that one table; the hostless `run` entry point installs an empty table, so such a program provably has no effect beyond the string it prints.

- **Bounded output and bounded nesting.** `print` writes through a byte budget

that clips at the cap without splitting a character and lets the run continue; the parser rides the kit depth guard so deep nesting is a diagnostic, never a stack overflow.

The cost model is exact, not approximate: it is a total function of the AST. The test `the_cost_model_is_exact` pins `print 1;` at exactly 2 fuel, ‘let x = 1 + 2; `at 4, and` repeat 3 { print 0; } `at 11` (cargo test -p prooflite‘). An agent generating `prooflite` can predict a program's cost before running it.

3.2 What fuel-bounded totality costs

Totality is not free, and the price is expressiveness. `prooflite` gives up general recursion and unbounded iteration entirely; a loop that does not know its own length is not expressible, because "run until a condition" is exactly the shape fuel exists to forbid. Arithmetic that would wrap around instead halts: overflow, division or remainder by zero, and negation of `i64::MIN` are each a diagnostic, never a silent wrong-but-clean result (`checked_arithmetic_never_wraps`, `cargo test`

`-p prooflite`). Even `i64::MIN` is not writable as a literal — `-` is an operator, so the positive half overflows first — and must be reached arithmetically.

This is the honest trade the paper's claim rests on. A general-purpose language buys `while`, closures, and unbounded recursion, and pays for them by making termination undecidable. `prooflite` refuses those constructs and, in exchange, makes termination a property the evaluator enforces on every input without analysis. Verification completeness and language size trade against each other, and this section is one qualitative illustration of that trade-off — not a rate measured across sizes: for a language whose target is agent-generated, mechanically selectable programs, the constructs given up are close to free and the guarantee bought is the one the selection loop needs.

3.3 The two crash-grade lessons: adversarial review is load-bearing

`prooflite` was built and then reviewed adversarially, and the review is where the interesting evidence is. It found two defects that the test suite as written did not, and both were process-killing, and both were failures of a *guarantee* rather than of ordinary logic — which is precisely the class a verification-first project cannot afford to ship. They are recorded in `GENESIS.md` under "post-genesis lessons."

The AST-spine stack overflow (M1). The parser rides `parselite`'s depth guard, so nesting is capped. But a left-associative operator chain like `1+1+...+1` folds *iteratively* in the parser — it costs O(1) guard entries — while building an AST whose left spine is as deep as the chain is long. The evaluator, and even the drop glue that frees the tree, later recurse down that spine. A flat 50,000-term source therefore parsed cleanly, consumed no suspicious amount of fuel, and then aborted the whole process on a stack the fuel budget could not touch. The lesson is sharp: **the depth guard bounds parser recursion, not AST depth.** The fix charges one guard entry per fold, so a chain's spine obeys the same cap as nesting, and reshapes else-if chains into a flat vector so the common flat program stays unbounded without deepening the tree. The invariant is now pinned by `operator_chains_count_toward_the_depth_cap`: a chain of 50 evaluates, a chain of 500 is an `E0102` diagnostic, and 5,000 folds parse-then-drop safely (`cargo test -p prooflite`).

The table-snapshot trust hole (M2). The host capability table is validated once, before the program runs, and that validated table is what the checker, the docs, and the parity manifest all vouch for. But the evaluator originally re-fetched `host.caps()` at every call site. A host with interior mutability could therefore serve one table to the validator and a different table to dispatch — the guarantee "calls resolve only against the vouched-for table" was a lie against an adversarial host. The fix fetches the table exactly once, snapshots the validated `Copy` value, and resolves the entire run against that snapshot; `the_validated_table_snapshot_drives_the_whole_run` asserts that `caps()` is called exactly once across a two-call program (`cargo test -p prooflite`). The general form, which recurs at M3 and M4: validate every channel into the artifact, then use the validated *value*, never a fresh fetch.

Neither defect was a typo. Each was a place where a stated guarantee — "bounded nesting never crashes," "the table is the complete, checked effect bound" — held for cooperative inputs and broke for adversarial ones. Ordinary tests, written by the same mind that wrote the code, exercised the cooperative case. The M1 review was run by six independent finders against three refuters and surfaced 16 raw findings, 13 confirmed, 2 of them crash-grade; the M2 review, same six-and-three shape, confirmed 16, including one crash-grade-adjacent hole in the parity hash (`GENESIS.md`, post-genesis lessons). This is the section's second measurement, and it is a claim about process, not luck: when the product *is* the guarantee, the guarantee has to be attacked on purpose, because the

failure modes that matter are exactly the ones that a well-behaved test program never reaches.

---

4. Construction cost: the Nth language on the kit

The kit's premise is that a purpose-sized language is cheap to build once the reusable kernel is paid for. This section measures that cost directly: the two languages built on `litelite` against the three the parent project (`localharness`) hand-rolled before the kernel was extracted. The comparison is suggestive rather than controlled — the caveats at the end are load-bearing — but the numbers point one way.

4.1 What is being measured

The parents each re-implemented a kernel — byte lexer, recursive-descent harness with a recursion guard, fuel, spanned diagnostics — and then a language on top. `litelite` writes that kernel once and reuses it. So the honest unit of "construction cost" for the Nth kit language is its *marginal* LOC and tests: the language-specific code, not a fresh kernel each time.

The shared kernel is small. The four base crates `diaglite`, `lexlite`, `parselite`, `fuellite` total 960 LOC (`bash scripts/caps.sh`: 252 + 290 + 238 + 180), and both kit languages draw on it for free — this is the amortization the repo actually demonstrates. Adding the capability layer and both emitters brings shared infrastructure to 3,651 LOC (`bash scripts/caps.sh`, summing all seven kit crates), but that fuller figure is aspirational for the emitters: `evmlite` (1,400 LOC) and `modlite` (770 LOC) are built and tested yet have no consumer in this repo, since both kit languages are tree-walk with no codegen. A future language that needs codegen *could* draw on them for free; none does today, so their reuse is claimed, not shown (§6.4).

4.2 The two kit languages vs the three parents

Language	Role	LOC	Tests	Kernel / codegen
Parents (hand-rolled, in `localharness`)				
`rustlite`	Rust subset → wasm	8,000	99	own kernel + wasm emitter
`soliditylite`	Solidity subset → EVM	7,600	159	own kernel + EVM assembler
`bashlite`	fuel-bounded shell	3,100	64	own kernel, **no depth guard**
On `litelite`				
`prooflite`	total reference language, tree-walk eval	1,955	31 + 2 doc	reuses kit kernel; no codegen
`stratlite` + `backtestlite`	total strategy language + its verifier	2,562	16 + 2 doc	reuses kit kernel; no codegen

Parent figures are quoted from `GENESIS.md` ("The lineage") and describe crates in a different repository; they are not reproducible by a command here. Kit-language LOC comes from `bash scripts/caps.sh`; kit-language test counts from `cargo test -p prooflite`, `cargo test -p stratlite`, and 'cargo test -p backtestlite (prooflite `31 unit + 2 doctests;` stratlite' 10 + 1; `backtestlite` 6 + 1; the last two sum to the 16 + 2 shown). The `stratlite + backtestlite` LOC is the sum of the two crates, 1,893 + 669, because M4 split one language across two crates — the trader's live dependency and the selection loop's verifier — under one evaluation path.

The marginal kit language lands at 1,955 and 2,562 LOC against parents that re-rolled everything at 3,100–8,000 LOC. The structural reason is the amortized kernel, not a smaller language per se — which is exactly why the scope caveat below matters.

One parent number is worth isolating: `bashlite`, the smallest at 3,100 LOC, shipped with *no* parser recursion guard (`GENESIS.md`, "What was ported at genesis") — a live stack-overflow bug in production. The kit makes that guard the only door into the parse harness (`parselite`, `DEFAULT_MAX_DEPTH = 96`), so the class cannot recur. Smaller hand-rolled did not mean safer; it meant a missing invariant nobody re-derived.

4.3 Wall-clock: one recorded session per language

The one-session claim traces to commit timestamps ('git log –format='%h|%ad|%s' –date=format:'%Y-%m-%d %H:%M:%S' a557fef..1524eb3', with the genesis commit timestamp from `git show -s a557fef`). The entire kit plus both languages plus both emitters landed in a single overnight span:

- Genesis (kernel crates): `a557fef`, 2026-07-14 22:10:31.
- `prooflite` (M1): landed `07c2877` at 2026-07-15 00:12:30; adversarial-review

fixes `9ac20b3` at 00:40:37 — a 28-minute review-and-fix window, with no commit between the genesis port snapshot (22:13:07) and the land.

- `stratlite` + `backtestlite` (M4): landed `42667a6` at 02:59:24; review fixes

`1524eb3` at 03:34:19 — a 35-minute review-and-fix window; the whole milestone (from M3 done at 02:25:10) spans 1 h 09 m.

- End to end, genesis through M4 review-complete: 22:10:31 → 03:34:19,

5 h 24 m, for four kernel crates, the capability layer, two emitters, and two languages.

Commit timestamps bracket elapsed time between commits; they include idle and cannot separate think-time from typing, so treat them as the recorded envelope of a session, not a stopwatch. What they do establish is that each language went from nothing to landed-and-adversarially-reviewed inside a single dated sitting, not the "~a week" the roadmap had budgeted (`GENESIS.md`, "The three questions").

4.4 Defects at landing

Each language shipped through the same adversarial review (multi-agent finder/refuter panel) and the counts are recorded in `GENESIS.md` ("Post-genesis lessons"): `prooflite` drew 16 raw findings →

13 confirmed, 2 crash-grade; `stratlite/backtestlite` drew 14 confirmed. Notably, `prooflite`'s two crash-grade defects were the *same* lesson — the parser depth guard bounds recursion, not AST depth, so a flat 50K-term source builds a spine the evaluator recurses down (§3.3) — and that fix was paid once, into the kit's consumption idiom (every iterative AST-deepening fold now charges the guard). These counts come from review panels recorded in `GENESIS.md`; they are not re-runnable and depend on panel composition, so they index review thoroughness, not an absolute defect density.

4.5 Caveats

The reading here is deliberately narrow.

- **N = 2.** Two kit languages is a data point, not a trend. The seam-tax

question — whether the kit is a net win or a tax a re-homed parent would refuse — stays open until M5 re-homes `bashlite` (`GENESIS.md`, "The three questions"); §6.4 returns to it.

- **Same developer, same model era.** Both kit languages and all three parents

were built by the same developer with the same model generation. There is no controlled A/B; this is not evidence that the kit helps a *different* builder.

- **LOC is a proxy** for effort and for scope. It is not normalized to language

power.

- **Scope mismatch cuts against the aggressive reading.** `prooflite` and

`stratlite` are tree-walk evaluators with no codegen; the compiling parents (`rustlite` → wasm, `soliditylite` → EVM) carry emitters the kit factors into `evmlite/modlite` (2,170 LOC combined) that neither kit language yet uses. A fair like-for-like — a kit language that actually emits code — is future work. Until then, the 1,955/2,562-vs-3,100–8,000 gap conflates the amortized kernel with a genuinely lighter language scope, and only the kernel amortization is on firm ground.

The defensible claim: building the Nth language on the kit costs its language-specific LOC only, because the kernel is paid once; the two kit languages landed at 1,955 and 2,562 LOC in single recorded git sessions, each surviving adversarial review, against parents that re-rolled their kernels at 3,100–8,000 LOC. Whether that advantage survives a codegen-scope language and a second builder is what M5 and later data points must answer.

---

5. Verified selection, a verifier-as-reward specification, and the fine-tune result

Sections 3 and 4 measured what smallness costs and what it buys at the language boundary. This section puts the guarantees to work: it uses the mechanical verifier as a selection filter over

agent-generated programs, and then specifies it as a training reward. The claim under test is narrow and mechanical — that a purpose-sized language turns "is this program acceptable?" into a total, deterministic, third-party-checkable predicate — and the results are reported with an equally narrow honesty about what the single-month benchmark can and cannot show.

The instrument shipped complete. The verifier-only GPU fine-tune has now been run twice — §5.6 reports it on `stratlite` and §5.7 its N = 2 replication on `prooflite`; the remaining experiment, a frozen-model A/B arm for which there is no API key, has not been run, and §5.9 marks it as a PENDING protocol slot with its exact commands, per the house rule that pending results are named, not promised.

5.1 The instrument: `verify()`, `equity_hash`, and the `Reject` histogram

The whole selection apparatus is one function. ‘backtestlite::verify(src, candles, limits, costs, gate)‘ compiles the source, backtests the strategy, and applies an activity gate, returning `Result<(Strategy, Report), Reject>` (`crates/backtestlite/src/lib.rs:199`). The error type is the selection signal itself:

```
enum Reject { Compile(Diag), Run(Diag), Gate(GateFail) }
```

Every rejected program lands in exactly one of three bins — the source is ill-formed (`Compile`), it faulted at run time or on the data (`Run`), or it ran clean but did not behave like a strategy (`Gate`). Tallying those bins over a pool is the *Reject histogram*: structured selection pressure, not a scalar pass/fail. The gate defaults to `min_trades = 4`, `min_bars_evaluated = 16` (`crates/backtestlite/src/lib.rs:89`).

Two properties make the predicate usable as an oracle rather than merely a checker:

- **Determinism as one number.** `Report::equity_hash` is an FNV-1a-64 hash of

the final equity curve, so an entire backtest reduces to a single reproducible integer; `Report` also derives `Eq`. The determinism is a pinned test (`determinism_is_exact_and_hashable`), so "the same strategy on the same candles scores identically" is enforced, not assumed (`cargo test -p backtestlite`, 7 tests pass).

- **No look-ahead by construction.** `stratlite` has no name for a future bar

and fills at the next open; this is a grammar fact, pinned by `no_lookahead_prefix_invariance`, which mutates every candle after each index `k` and asserts no decision at or before `k` changes (`crates/stratlite/src/lib.rs:256`; `cargo test -p stratlite`, 11 tests pass).

Because evaluation is fuel-bounded (`Limits::default().fuel_per_bar = 25_000`, `crates/stratlite/src/lib.r`
computing a verdict is itself guaranteed to terminate — the property that lets the same predicate serve as a training reward in §5.5 without any generated program ever hanging the loop. The instrument is 1,893 LOC of language plus 669 LOC of verifier (`bash scripts/caps.sh`), zero-dependency, and the generation prompt is `stratlite::REFERENCE` — a `const` of the crate (`crates/stratlite/src/lib.rs:78`), so the language the model is shown and the language the verifier enforces cannot drift.

5.2 The key-free corpus run: real numbers

Running §5 end-to-end against a frozen API model needs an API key. To produce real numbers without one, the generation step was performed by six agents — one per strategy family (trend, breakout, mean-reversion, momentum, stateful, combo) — each handed `stratlite::REFERENCE` and asked for diverse valid programs. The result is committed as `experiment/corpus/seed.jsonl`: 134 programs, `{id, style, source}` per line, balanced across the six families (trend 23, breakout 23, mean-reversion 22, momentum 22, stateful 22, combo 22; `wc -l experiment/corpus/seed.jsonl` and a one-line `Counter` over the `style` field). This is generate→verify→keep with agents as the generator; the honest split holds — the *generation* does not reproduce and so is committed rather than regenerated, while every number below is a pure function of that committed corpus plus the pinned candles.

The candle data is `BTCUSDT-1h-2024-01.csv`: 744 hourly bars, chronologically split 60/40 into a 446-bar train window and a 298-bar held-out window, with adverse costs (5 bps fee, 1 bps slippage) fit to the train window only — `fee = 2196`, `slip = 439` ticks, held-out/train price drift 0.943x (`cd experiment && cargo run -q - data data/BTCUSDT-1h-2024-01.csv`). Verifying the corpus against the train window (‘cd experiment && cargo run -q – reward corpus/seed.jsonl data/BTCUSDT-1h-2024-01.csv‘) gives the Reject histogram:

Outcome	**Count**	**Rate**
Compile-fail	0	0%
Run-fault	0	0%
Gate-fail	2	1.5%
Survivor (`ok`)	132	98.5%

Compile rate is 100% and the survivor rate is 132/134 = **98.5%**. The two non-survivors are gate failures, not faults: they parsed and ran cleanly but did not trade enough to count as strategies. Strong agents, asked for valid programs and given the language card, can produce valid active `stratlite` almost every time; this measures conditioned generation, not an unconditioned base rate — the calibrating comparison is the PENDING, permanently-keyless A/B arm (§5.9), so we do not read a base rate off this number. It is also the setup for the limitation in §5.4. The same run reports 134 distinct source-canonical novelty keys over the 134 programs, so no two are template clones under the dedup key of §5.5.

5.3 The fuel bound is free on this task — measured, not assumed

The termination guarantee is only interesting if we can say whether it *bound*. Across the 132 survivors, `max_fuel_per_bar` is **min 16 / median 55 / max 186** against the 25,000/bar cap — the single most expensive diverse strategy consumes **0.74% of the budget** (same `reward` command; the field is emitted per line). So on this task the fuel bound never came close to binding: it is free, and, as a *discriminator*, untested. The guarantee is genuine — any program that looped would be cut at 25,000 — but this corpus never exercised it. §6.1 develops this as a named negative result about our own evidence.

5.4 The conditional metric and the no-teeth finding

The naive way to compare generators is raw held-out survivor rate. It is misleading, and identifying exactly why is a methodological contribution of this section rather than an afterthought.

The `Compile` rung of the histogram is **data-independent**: whether a program parses (and largely whether it halts) does not depend on the candles at all — a program that compiles on train compiles identically on held-out. So a raw "held-out survivor lift" can be almost entirely a compile-rate lift that shows up identically on train: grammar-learning dressed as generalization. The honest metric therefore *conditions on compiling* and reports the **gate-clear rate** (the genuinely data-dependent rung) on each window, plus the train-minus-held-out gap.

Running that metric (‘cd experiment && cargo run -q – eval corpus/seed.jsonl data/BTCUSDT-1h-2024-01.csv‘):

- compile rate **100%**;
- among compilers, gate-clear **98.5% train** vs **97.0% held-out**;
- **gap = 1.5 points — near zero.**

A near-zero gap is a finding, and we report it as one rather than hiding it: on one month of one asset, the held-out window is no harder than the train window, so raw survivor rate carries essentially no out-of-sample signal here. The per-style breakdown confirms the gap is not an aggregate artifact — five of six families clear at 96–100% held-out, and only mean-reversion shows any train/test spread (19/22 = 86%):

Style	**Held-out gate-clear (among that style's compilers)**
breakout	100% (23/23)
combo	100% (22/22)
momentum	100% (22/22)
stateful	100% (22/22)
trend	96% (22/23)
mean-reversion	86% (19/22)

The methodological consequence is a precondition on any future claim: **prove the benchmark has out-of-sample teeth before claiming a lift on it.** A wider regime split — a chronologically distant test window and a second asset, chosen so that a trend-fitted strategy actually goes silent — is a prerequisite for the generalization experiment, not a nice-to-have. The instrument surfaces its own benchmark's inadequacy, which is the behavior we want from a verifier; §6.2 draws out the consequence for the fine-tune benchmark.

5.5 Verifier-as-reward (M6): the ladder and the anti-hacking guards

§5.2–5.4 select from a fixed pool. M6 moves one knob: it puts the model in the loop and uses the same verifier as the training reward. M6 tests whether the tempo-x402 result (§1) would generalize off `rustc` onto a purpose-sized language — the hypothesis the §5.6 and §5.7 runs now test.

Three properties of a fuel-bounded verifier make it a reward oracle `rustc` cannot be: computing a rollout's reward always terminates (no generated program can hang training), totality removes reward-hacking-by-nontermination as a category, and the reward is CPU-cheap and deterministic (a checkpoint's rewards reproduce from a command even though the model that produced the rollouts never can).

The reward is a four-rung validity ladder read directly off `Reject` (`experiment/src/reward.rs`): `Compile` → `0`, `Run` → `1/3`, `Gate` → `2/3`, `Ok` → `1`. **Train PnL is deliberately excluded from the reward value**, for two anti-hacking reasons: rewarding it would make the held-out comparison circular (you would be measuring what you optimized), and continuous train PnL is the single most overfittable signal on a few months of one asset. Excluding PnL does not, however, remove all overlap between what the reward optimizes and what the win condition scores: the reward rewards the gate-clear (`Ok`) rung, which is also exactly what the §5.6 benchmark measures on held-out data. This is standard train/test discipline only insofar as the held-out window is genuinely harder than train — a precondition §6.2 shows the single-month benchmark fails — so a gate-clear lift, even on the wider regime split §5.6 uses, reads as grammar competence rather than out-of-sample edge. PnL, fuel, `equity_hash`, and a dedup key are all *emitted* so a trainer can reshape and own that choice, but `value == ladder(class)` is a pinned invariant. This is the yield/edge split: M6 is designed to yield a better *generator* (validity and diversity of the pool) — §5.6 reports that it does, and §5.7 replicates it on a second language; finding the profitable strategy stays §5.2's *selector*. A validity-only reward means a null PnL result falsifies nothing and cannot be spun as success.

The red team named five reward hacks; each has a guard, and the guards are tested with no GPU:

1. **Empty/junk rollouts.** An empty source or a fenced code block scores a

compile-zero, not a lenient partial — verified as given, never repaired (`the_empty_rollout_hack_is_closed`, `garbage_is_a_compile_zero_not_a_leniency`).

1. **Gate-rung farming.** `lookback 4;` compiles, runs clean, gate-fails, and

collects 2/3 for the *absence* of a strategy. Guard: SFT admission keeps Ok-rung survivors only; the 2/3 rung is a diagnostic and a GRPO signal, never an admission threshold (`experiment/train/admission.py`, `test_only_ok_rung_is_admitted`).

1. **Template mode-collapse.** Guard: `novelty_key`, a source-canonical

FNV-1a-64 over comment-stripped, whitespace-collapsed source, caps any one key's share of the SFT set. It sees through re-commenting and whitespace-run reformatting but — stated as an honest limit — not token-adjacency spacing (`x=1` vs `x = 1`), constant-perturbation, or semantic clones (`novelty_key_ignores_comments_and_whitespace_but_not_logic`, `test_dedup_caps_a_template_family`).

1. **`equity_hash` diversity gaming.** A one-tick constant change yields a new

curve and a new hash. Guard: use `novelty_key`, which is defined below the survivor rung where `equity_hash` is 0 for everything, for both dedup and the diversity metric.

1. **Easiest-style collapse.** Guard: a per-style admission cap keeps the set

spread across families (`test_per_style_cap_keeps_the_set_spread`).

The Rust reward core and its guards are covered by `cd experiment && cargo test` (23 tests pass), and the Python admission guards by 'cd experiment/train && python3 test_select.py' (6 pass) — both stdlib-only, both runnable now. The Rust/Python seam is exactly two things: the reward CLI (candidate JSONL in, reward records out) and the generation format. Rust owns verify, the reward scalar, the histogram, fuel, and the dedup key — all reproducible today; Python owns sampling and the weight update, which need a GPU and do not run in CI. The trainer lives outside the kit's workspace (it takes `torch`) with its own cap counter; `experiment/` is 1,489 LOC, `experiment/train/` is 639, and the N=2 reward tool `experiment/proofbench/` is 354 (`bash scripts/caps.sh`).

5.6 The verifier-only fine-tune (M6): from a measured-zero floor to competent

The M6 instrument has been run. A small open-weights model, the dense `Qwen/Qwen3-0.6B` (`experiment/train/train.py:55`), was fine-tuned to generate `stratlite` using only the kit's verifier as supervision — no teacher model, no API key. Cold-start SFT on the 132 committed corpus survivors (§5.2) was followed by eight rounds of verifier-only rejection-sampling self-play: sample per style from checkpoint `C_{r-1}`, batch-score with the reward CLI, admit `Ok`-rung survivors deduped by `novelty_key` under per-style caps (§5.5), SFT to `C_r`, repeat. On the train reward window the full-survivor rate climbed monotonically — cold-start → R0 39.0% → R1 67.0% → R2 80.6% → R3 85.8% → R4 89.6% → R5 94.6% → R6 96.0% → R7 98.2% (`experiment/results/train_curve.log`) — and training stopped at checkpoint C7 on the protocol's train-saturation early-stop (rich-rate). `distinct_nkeys` rose then narrowed — 392 → 624 → 685 → 654 → 694 (peak, R4) → 674 → 555 → 534, over 399–809 admitted per round — the same peak-then-narrow shape §5.7 later finds for prooflite and §6.6 treats as a general limit. The anti-collapse guards of §5.5 held the model to a diverse grammar rather than one template through the rising limb, but did not prevent the post-peak narrowing; because C7 was selected on rich-rate saturation, not the distinct-key peak, it sits past its diversity optimum (~R4). That does not touch the gate-clear result below — a validity claim, and validity is monotone — but it does mean C7's generator is less varied than an earlier checkpoint's would be. A held-out novelty check confirms the model learned the grammar, not the corpus: of C7's 251 gate-clearing programs, 250 (99.6%) carry a source-canonical key absent from the committed corpus's 134 keys (205 of them distinct; `s5 reward` emits the key per program), the same anti-memorization control §5.7 runs for prooflite — now symmetric across both arms. Fuel over survivors stayed at ~1% of the 25,000-per-bar cap throughout, as in the corpus run (§5.3) — a stratlite-specific figure §6.1 revisits against prooflite.

C7 was benchmarked against the same model before fine-tuning, with identical non-thinking prompts (256 samples each, 32 per style over 8 styles), scored by the deterministic verifier on three windows: the BTC January 2024 train reward window, a distant BTC June 2024 window under a five-month embargo, and an ETH June 2024 window on an asset never trained on. The conditional metric of §5.4 — compile rate over the pool, then gate-clear among compilers — reads:

window	base compile / gate-clear	C7 compile / gate-clear
BTCUSDT Jan 2024 (train reward window)	0.0% / 0.0%	100.0% / 95.7%
BTCUSDT Jun 2024 (distant, 5-month embargo)	0.0% / 0.0%	100.0% / 96.5%

window	base compile / gate-clear	C7 compile / gate-clear
ETHUSDT Jun 2024 (cross-asset, never trained on)	0.0% / 0.0%	100.0% / 96.1%

(‘cd experiment && ./target/release/s5 eval results/{base,c7}.jsonl data/<window>.csv; `full output in` experiment/results/benchmark.txt‘.) The baseline is a true floor, not a weak one: `stratlite` exists in no pretraining corpus, so base `Qwen3-0.6B` produces zero valid programs across 256 attempts on every window — it parrots the grammar card's notation but cannot emit one compiling program. The entire base→C7 lift is therefore the fine-tune, with nothing to confound that gap. This shares the RECIPE of the tempo-x402 result of §1 — compiler-verified self-play, no teacher — but measures a different variable: tempo-x402's 1.5% → 16.4% is pass@1 at SOLVING 201 specified Rust problems, a language in pretraining, whereas this is the VALIDITY rate of open-ended generation of a language absent from it. The measured-zero floor reflects that absence — the grammar must be acquired from nothing — not a task harder than solving Rust; what carries across is the recipe, a fuel-bounded verifier standing in for `rustc` as the oracle on a language no compiler corpus ever saw.

What this establishes is bounded exactly as §5.5 designed it. Verifier-only fine-tuning takes a small model from no competence to ~96% valid-strategy generation on a language it never saw pretrained, and that competence holds across a five-month embargo and across assets (95.7% / 96.5% / 96.1% held-out gate-clear); the per-style held-out breakdown on the cross-asset window is near-uniform (88–100%), so no family collapsed onto one easy template. It does **not** establish edge. The verifier certifies well-formed and active, never profitable, and the train-minus-held-out gap stays near zero (2.3 / 1.2 / 1.6 points). Read against §6.2, that near-zero gap is the honest signature of grammar competence — the language is genuinely as easy on ETH June as on BTC January — not of generalizing skill at finding good strategies. Selecting the profitable strategy from this now-competent generator stays the §5.2 selector's job (`pick_verified`), not the generator's; because the reward is validity-only (§5.5), this result carries no edge claim and cannot be spun into one. The MODEL does not reproduce — sampling is stochastic and a fine-tune is not bit-identical across hardware — but the SCORING does: the two sample pools (`results/base.jsonl`, `results/c7.jsonl`, 256 programs each) and the candles are committed, and the `s5 eval` command above reproduces every number in the table.

5.7 The second-language replication (M6, N = 2): the same reward on a data-free language

If the §5.6 lift were an artifact of one grammar it would not transfer, so the instrument was run a second time on `prooflite` (§3) — the reference *compute* language, which reads no market data, places no trades, and has no held-out window. The language-parametric trainer ran UNCHANGED; only the reward binary was swapped (`s5` → the parallel `p6`, `experiment/proofbench/src/main.rs`), which serves the prooflite prompt card, eight computation-family styles, and the same validity ladder — `compile` → `run` → `gate` → `ok`, where the `ok`/RICH rung requires a clean run that prints ≥ 3 distinct lines over ≥ 30 fuel. Cold start on the 174 committed corpus survivors — the `ok`-rung keepers of 175 key-free agent drafts (99.4% RICH raw; the survivor file `experiment/proofbench/corpus/seed.jsonl` is 100% RICH under `p6` by construction) — then nine rounds of the identical rejection-sampling self-play. Rich-rate climbed monotonically — cold-start → R0 43.8% → R1 63.5% → R2 72.4% → R3 83.1% → R4 85.7% → R5 90.7% → R6 91.2%

→ R7 95.0% → R8 96.2% (of 1,024 sampled per round).

Because prooflite reads no data there is no train/held-out DATA split, so the generalization question changes shape: are the rich programs LEARNED, or MEMORIZED from the 174 human-authored cold-start examples — the only external data the model ever saw? The `p6 novelty` command answers it by source-canonical key (FNV-1a-64 over comment-stripped, whitespace-collapsed source, `main.rs:novelty_key`, which sees through format and comment clones). Benchmarking the selected checkpoint against the base model, identical non-thinking prompts, 256 samples each (32 per style):

model	parse	RICH (ok)	distinct rich keys	novel ok / ok (∉ corpus)
base `Qwen3-0.6B`	23.4%	3.5%	9	9 / 9 (100%)
C5	96.9%	90.6%	213	232 / 232 (100%)
C6 (selected)	99.2%	94.5%	**216**	242 / 242 (100%)
C7	100.0%	96.1%	199	245 / 246 (99.6%)
C8	98.8%	96.1%	205	245 / 246 (99.6%)

(`cd experiment/proofbench && ./target/release/p6 eval results/c6.jsonl` and `./target/release/p6 novelty results/c6.jsonl corpus/seed.jsonl`; full output in `experiment/proofbench/results/benchmark.t`
The floor is again true: prooflite is in no pretraining corpus, so base `Qwen3-0.6B` — which recognizes the C-like surface enough to parse 23.4% of its attempts, more than stratlite's 0% — still writes a RICH program only 3.5% of the time (9 of 256). The lift from 3.5% to ~95% is the fine-tune, and ~100% of the rich programs are novel against the corpus — so the competence is not recall of the 174 human examples, the only external text the model saw. (Novelty is measured against that human seed only, not against the model's own admitted self-play programs, so it rules out memorizing the human corpus, not reproduction from the larger self-generated training set.)

The selected checkpoint is C6, and why is itself a finding. Raw validity saturates across C6–C8 (94.5% / 96.1% / 96.1% RICH), but DIVERSITY does not: admitted distinct keys peak at round 6 (823) then fall monotonically (750, 686). C6 also holds the most distinct rich programs on the held-out benchmark (216, against C5's 213, C7's 199, and C8's 205), so selecting it is defensible on both measures — but the benchmark is a single 256-sample draw per checkpoint with no variance estimate: C5's 213 sits within a hair of C6's 216, and the C7-vs-C8 order (205 > 199) inverts the training curve's (750 > 686), so around the peak the draws disagree within sampling noise. The load-bearing evidence is therefore the training-curve decline, not the benchmark: past round 6 the policy trades breadth for reward — mild mode-narrowing that the anti-collapse guards of §5.5 bound but do not abolish — so the method has an optimal stop visible in the distinct-key curve rather than the rich-rate. What this replication establishes is bounded: verifier-only fine-tuning is not a ONE-grammar artifact — run unchanged on a language with no data, no market, and no trades, only checked arithmetic and bounded loops, it takes a small model from 3.5% to ~95% rich generation with ~100% corpus-novelty. It does not establish generality past the confounds both arms share — same kit and author (as for the construction claim, §4.5) and, specific to the fine-tune, the same base `Qwen3-0.6B`, trainer, and validity-ladder reward shape; generality across models in particular is untested. What it also does not establish is that every rich program is INTERESTING; the RICH rung certifies well-formed, terminating, and varied output, and selecting genuinely useful programs from this competent generator stays a downstream concern, exactly as edge does for stratlite. As in §5.6 the MODEL does not reproduce, but the SCORING does, from the committed pools.

5.8 Transfer: solving held-out problems, and what self-play optimizes

The abstract flags the gap honestly: §5.6–5.7 measure generation VALIDITY, where tempo-x402 measured task-solving pass@1. This section closes that gap with a held-out problem-solving benchmark on `prooflite` (`experiment/proofbench/problems/heldout.jsonl`): 30 specs in three tiers (9 easy / 9 medium / 12 hard — the hard tier needs real algorithms: primality, popcount, digit-sum, Collatz length), each with a verified reference solution. A candidate SOLVES a problem iff its output exactly equals the reference's; 8 samples per problem per model, and the pools are committed (`results/solve_{base,cinit,c6}.jsonl`), so 'p6 solve problems/heldout.jsonl results/solve_<m>.jsonl' reproduces every number. Three arms separate the recipe's ingredients: **base** `Qwen3-0.6B`; **Cinit**, the cold-start checkpoint — supervised fine-tuning on the 174 human corpus programs and nothing else, i.e. the plain-SFT baseline; and **C6**, Cinit plus six rounds of verifier-only self-play — the checkpoint §5.7 selects on the generation benchmark. (Cinit's own generation row, same 256-sample protocol: 83.2% parse, 48.8% RICH, 124 distinct rich keys — between base and the self-play curve, as expected.)

model	RICH generation	solve pass@8	easy	medium	hard	pad
base	3.5%	20.0% (6/30)	3/9	2/9	1/12	1
Cinit (plain SFT)	48.8%	**80.0%** (24/30)	9/9	8/9	7/12	1
C6 (+ self-play)	**94.5%**	43.3% (13/30)	6/9	5/9	2/12	8

Two findings, one expected and one not — and the second is the sharper.

Transfer is real. Training on verifier-selected programs — never on problem/solution pairs — takes pass@8 from 20% to 80%, including 7 of the 12 hard discriminators. Base solves a few easy problems from surface familiarity alone (the C-like syntax parses often enough for counting loops to land), so the floor is not zero here; the lift is what carries the tempo-x402 shape onto this instrument, now in its original task-solving units.

Self-play optimizes the verifier, and the verifier is not the task. The checkpoint that wins generation (C6, 94.5% RICH against Cinit's 48.8%) loses nearly half of Cinit's solving (43.3% against 80.0%). The `pad` column is the mechanism: `p6 solve` separately reports candidates whose output has the reference answer as a strict prefix followed by extra output — the right algorithm, then spurious prints. Eight of C6's seventeen misses are exactly that (against one each for base and Cinit): a correct `sum_1_100` loop followed by a dozen filler `print` statements is the RICH rung ($\geq$3 distinct output lines over $\geq$30 fuel) speaking, distilled by self-play into an unconditional output habit that overrides the spec. The localization is exact. On the 12 problems whose correct output is itself RICH-shaped ($\geq$3 distinct lines), C6 matches Cinit — 9/12 versus 10/12; on the 18 problems whose correct output is short — where the training reward actively disprefers the correct shape — Cinit solves 14/18 while C6 solves 4/18 and pads 8 of the remaining

1. Where reward and task agree, self-play costs nothing measurable; where

they conflict, the policy obeys its internalized reward, not the prompt.

The two capabilities therefore move in OPPOSITE directions under self-play — generation validity 3.5 $\rightarrow$ 48.8 $\rightarrow$ 94.5, spec-conditioned solving 20 $\rightarrow$ 80 $\rightarrow$ 43 — and the diversity narrowing of §6.6

has a sharper companion: the policy does not merely narrow, it carries the reward's preferences into contexts the reward never saw. This is the Goodhart argument of §6.3 measured at the policy level, and it revises the recipe's reading. Cold-start SFT buys the language AND the ability to use it on demand; self-play buys the verifier's rungs, specifically — run it only while its rungs and the downstream task agree, and read the pad diagnostic as the early warning. The caveats are the section's own: pass@8 over 30 problems is a coarse single-draw instrument; one base model, one language; and the prefix-shaped `pad` count is a lower bound on "right answer, wrong shape" (answer-inside-padding is not counted). Solving was also never in the training objective for EITHER checkpoint — the surprise is not that C6 solves worse than a solver would, but that plain SFT solves this well and self-play then gives half of it back.

If that diagnosis is right — the deficit is the reward's shape, not self-play itself — then changing what the verifier rewards, and nothing else, should move it. A fourth arm tests exactly that. Same trainer, same admission guards, same Cinit start; every sampling prompt is now a training task ("a program that {spec}") drawn from 48 training problems with verified references (`problems/train.jsonl` — ids disjoint from the 30 benchmark problems, and with NO family overlap with the held-out hard tier: no primality, popcount, digit-sum or -reverse, Collatz length, Fibonacci, powers, or square/cube/odd sums); and the reward's top rung is CORRECTNESS — exact output match against that spec's reference (`p6 solvereward`) — instead of RICH shape. Correct-of-768 on the training specs climbed 27.0% → 64.3% → 81.3% → 88.7% → 93.0% → 93.1% over six rounds (`results/train_curve_solve.log`), and on the held-out benchmark:

model	**reward trained on**	**RICH generation**	**solve pass@8**	**hard tier**	**pad**
Cinit	(none — plain SFT)	48.8%	80.0%	7/12	1
C6	RICH shape	**94.5%**	43.3%	2/12	8
S5	spec correctness	24.2%	**93.3%**	**10/12**	1

The fix works, and overshoots: S5 solves 28/30 — easy and medium perfect, and the hard tier rises from Cinit's 7/12 to 10/12 even though the training specs share no family with it (alternating sums, primality, Fibonacci newly solved) — so the gain is composition transferring across task families, not leakage; the train-vs-held-out gap is small but real (97.9% on the 48 training specs vs 93.3% held-out). One round of correctness self-play (S1, 86.7%) already beats plain SFT, and the padding habit is gone (one `pad` against C6's eight). And the mirror image completes the argument: S5's unconditional generation drops to 24.2% RICH (parse 91.0% — the programs are short, terse, correct-shaped) against C6's 94.5% and Cinit's 48.8%. Each self-play arm maximized exactly its own reward and degraded the other axis relative to Cinit. Self-play does not make the model generically better; it makes the model more like its reward, everywhere, including in contexts the reward never scored. Which capability you get is decided by which predicate you hand the verifier — the design freedom §5.5 claims for verifier-as-reward is real, and it cuts both ways. Scoring for all four arms reproduces from the committed pools (`solve_s{1,3,5}.jsonl`, `solve_s5_train.jsonl`, `s5gen.jsonl`).

A third language then applied the lesson as day-one design rather than a retrofit. `applite` (§2's kit, a total UI-app language: state, widgets, fuel-bounded atomic event handlers) shipped with a reward whose top rung is BEHAVIORAL — a task is a spec plus an event script (clicks by button label, typing, assertions over rendered labels; `experiment/appbench`), so a valid app that does the

wrong thing scores as wrong by construction. One hour of the same trainer (cold start on 120 script-validated corpus programs, six spec-conditioned rounds) took the same base model from **0/16** held-out app specs — all 128 attempts compile-fail; the language predates the run by a day, an exactly-zero floor — to **16/16** behavioral pass@8, 107/128 attempts passing their script outright (‘a8 eval tasks/heldout.jsonl results/solve_{base,c5}.jsonl‘). The bound stays stated: sixteen tasks, whose interaction grammar deliberately matches training’s (fresh combinations, not fresh genres). What N=3 adds over N=2 is not another validity number but the recipe’s intended form working end to end: a language, its behavioral verifier, and a one-evening local model that writes programs which provably halt AND demonstrably do what was asked — the generate→verify→keep loop as a running product (`app/`), not a benchmark.

5.9 PENDING (permanently keyless): the frozen-model A/B arm

One experiment on the instrument has complete plumbing and a committed protocol but has not been run, and cannot be: the frozen-model A/B arm. The §5 selection comparison — arm V (verified pick: among survivors, best train PnL, re-scored on held-out) versus arm U1 (the naive user: ask once, ship candidate 0 unverified) — consumes a batch generated by a frozen API model (`claude-opus-4-8`, no temperature/seed, so the batch is recorded not reproduced; `experiment/src/api.rs:21`). Both arms consume the identical pool, so best-of-N optimism is symmetric and cancels in the paired difference. This arm has **no API key and will not get one** (`experiment/src/api.rs:69`); the selection logic and its scoring are shipped and tested (`score::pick_verified`, `score::score_heldout`, `cd experiment && cargo test`), and the §5.2 agent corpus is the key-free stand-in for the generation step.

- Generate: `cd experiment && cargo run - submit <n> batch.json` then

`cargo run - poll <batch_id> raw.jsonl` (needs `ANTHROPIC_API_KEY`)

- Score the two arms (pure, no network):

```
cd experiment && cargo run -q - score raw.jsonl data/BTCUSDT-1h-2024-01.csv
```

The reproducibility split is exact and stated once: the deterministic verifier plus committed artifacts reproduce every number in §5.2–5.8 from a command — including both fine-tunes’ scoring (stratlite’s held-out gate-clear from committed candles, prooflite’s RICH-rate, novelty, and solve pass@8 from committed pools and corpus); the generation itself — agents, the frozen API model, or either fine-tune — does not, and is committed or recorded rather than regenerated.

---

6. Limits and negative results

The thesis of this paper is that smallness buys mechanical guarantees. This section reports where those guarantees did not bind, where they can be gamed, and what they provably cannot certify. Each finding is measured, not argued, and every figure below reproduces from a command in the repository.

6.1 On this task the fuel bound was free — which means it was not tested

The termination guarantee is the kit's headline: a `stratlite` program provably halts within its per-bar fuel, so computing its backtest — or its training reward — can never hang. On the corpus we can run, the guarantee never came close to binding. Across the 132 survivors of the 134-program corpus, per-bar fuel consumption ranges from a minimum of 16 to a median of 55 to a maximum of 186, against a default cap of 25,000 fuel per bar (§5.3; the field is emitted by 'cd experiment && cargo run -q – reward corpus/seed.jsonl data/BTCUSDT-1h-2024-01.csv `over` class:"ok"' rows; the cap is `Limits::default().fuel_per_bar` at `crates/stratlite/src/lib.rs:137`). The most expensive strategy any of six diverse agent families produced spends 0.74% of the budget.

This is a negative result about our own evidence, and it must be stated precisely. It does **not** falsify the termination guarantee: the guarantee is a proof about the evaluator, not a claim about the corpus, and it holds whether or not any program approaches the cap. What it falsifies is any claim that this corpus *exercised* the bound. As a proof the fuel cap is sound; as a discriminator — a filter that rejects programs — it did zero work here, so its discriminating power is untested. The honest artifact is therefore the full distribution (min 16 / median 55 / max 186), not the count of fuel-exhaustion faults, because a bare "0 programs exhausted fuel" cannot distinguish "the bound did its job" from "the bound was decoration." A regime that actually pressures the cap (deeper indicator windows, adversarial rather than cooperative generation) remains future work; until then the strongest claim the data supports is that fuel is cheap insurance whose premium, on cooperative inputs, is zero. That is task-specific, though: the N=2 `prooflite` fine-tune (§5.7) pushed measured fuel use, among its `ok`-rung programs, to a maximum of 55,756 of the 100,000-fuel default — 55.8% of budget, against stratlite's 0.74%, some 75× the budget fraction and far nearer the cap. On a compute language with real loops the bound came far closer to binding, so "the fuel bound was free" is a fact about the stratlite backtest, not about the kit.

6.2 The single-month benchmark has no out-of-sample teeth

The §5.4 selection experiment needs a held-out window that is genuinely harder than the training window; otherwise a "survivor lift" measures grammar-learning, not generalization. On the one month of hourly BTCUSDT candles we ship, the held-out window is not harder. Among the programs that compile, the gate-clear rate is 98.5% on train and 97.0% on held-out — a gap of 1.5 points — and per-style only mean-reversion shows any spread (86% held-out versus 100% for four of the other five families; 'cd experiment && cargo run -q – eval corpus/seed.jsonl data/BTCUSDT-1h-2024-01.csv', full table in §5.4). The aggregate gap is near noise.

This was predicted before it was measured. The compile rung is data-independent — whether a program parses does not depend on the candles — so a raw held-out survivor rate can rise entirely on grammar competence that shows identically on train. The eval metric is therefore deliberately *conditional* (gate-clear among compilers) rather than raw, and the pre-registered discipline (`experiment/M6.md`) requires proving the benchmark has out-of-sample teeth before any lift on it is claimed. The measurement says it does not, on one month of one asset. The consequence shaped the M6 fine-tune benchmark (§5.6): rather than score the fine-tune on this single month, we built the wider regime split this section demands — a chronologically distant BTC window under a five-month embargo and a second asset, ETH, never trained on. On that split C7's held-out gate-clear generalizes (95.7% / 96.5% / 96.1%), yet the train-minus-held-out gap stays near zero (2.3 / 1.2 / 1.6 points). The wider split thus confirms that what generalizes is the *language* —

grammar competence is regime- and asset-independent — and still shows no out-of-sample edge teeth, because the gap that would carry an edge claim is exactly the gap that stays near zero. We therefore report a competence lift off a measured-zero floor (§5.6), not an out-of-sample edge result.

6.3 Goodhart risk: hard caps push complexity into the seams

The constitution's LOC caps ($\leq$2,000 per crate, $\leq$25,000 repo) are mechanical and CI-enforced, and mechanical targets invite Goodhart's law: complexity that cannot live inside a capped crate migrates to wherever the counter cannot see it. The natural escape hatch is the experiment harness, which sits outside the kit's Cargo workspace precisely because it is allowed dependencies the kit forbids. If the caps counted only kit crates, the harness would be an unmetered sink into which any inconvenient logic could be pushed while every per-crate number stayed green.

The mitigation is to extend the counter across the seam rather than trust a boundary. `scripts/caps.sh` meters `experiment/src` at a 1,500-LOC cap (currently 1,489), the Python trainer at an 800-LOC cap (currently 639), and the N=2 reward tool `experiment/proofbench/src` at a 1,500-LOC cap (currently 354), alongside the kit's per-crate 2,000 and repo 8,214 / 25,000 (`bash scripts/caps.sh`). The harness cap sits 11 lines below its ceiling, which is itself a signal: the counter is close enough to bite, so it is not decorative. This does not eliminate Goodhart pressure — a determined author could still relocate complexity into a text corpus, a data file, or prose the LOC counter ignores — but it closes the one seam the architecture most invites, and it makes the cost of the seam visible in the same report as the caps it protects. The general limit stands: a mechanical cap disciplines the thing it counts and nothing else, and every uncounted surface is a place the discipline does not reach.

6.4 The seam-tax question is open; M5 has not answered it

The whole kit rests on a claim reality has not yet tested: that extracting the kernel three parent languages hand-rolled is a net simplification for a real consumer, not a seam-tax that makes every language pay an integration cost larger than the duplication it removes. The four milestones on the kit (`prooflite`, `caplite`, the two emitters, `stratlite`+`backtestlite`) show the kit *composes*, but every one of them was built to consume the kit — they cannot be disinterested witnesses. The two emitters are the sharpest instance: 2,170 LOC built and tested with no consumer anywhere in this repo, whose named consumer is precisely this pending re-homing, so their reuse payoff is argued from the parents rather than shown. The designed test is M5: re-home the parent `bashlite` onto the kit inside `localharness` and require the migration to shed net LOC there, or it does not ship. That milestone has not been run. Until it is, the load-bearing claim of the paper — that the kernel carries its weight for a consumer that did not exist to justify it — is supported by construction convenience and argument, not by an independent measurement. We name this as the single largest unverified claim in the work, not as a promise that it will come back positive.

6.5 Reward hacks in our own reward: the adversarial frame is necessary, not rhetorical

M6 uses the verifier as a training reward, and building it surfaced concrete ways a model could farm reward without producing a strategy (§5.5 enumerates the five hacks and their guards). What matters for the limits discussion is that these were found in our own oracle, which is the

evidence that the adversarial framing earns its keep rather than decorating the paper. Two are closed and tested with no GPU — the empty-rollout hack (an empty source now scores a compile-class zero, `the_empty_rollout_hack_is_closed`, one of eight tests matching `cd experiment && cargo test`) and the gate-rung hack (`lookback 4;` collects 2/3 for the absence of a strategy, so admission keeps Ok-rung survivors only, `test_only_ok_rung_is_admitted`, in the six-test stdlib-only trainer suite `cd experiment/train && python3 test_select.py`). Related hacks — `equity_hash` diversity gamed by a one-tick perturbation, mode collapse onto one template — drove the switch to a source-canonical `novelty_key` with a documented honest limit (it catches comment and format clones, not constant-perturbation or semantic clones).

The general finding: a reward oracle is an adversary's target, and the failures were not exotic — they were the cheapest programs in the language. That they existed in a verifier we built deliberately, and were found only by attacking it, is the concrete case for treating verification as physics across an adversarial boundary rather than as a checklist. It also bounds the claim: the guards close the hacks we found. We do not claim to have enumerated the reward's attack surface, and AST-level canonicalization — the refinement that would close the semantic-clone class — is unbuilt.

6.6 The generator narrows past its diversity peak, and two untested edges

The M6 fine-tune carries a negative result of its own, visible as a pattern rather than a fluke only because N=2 showed it twice. In both arms the admitted programs' distinct-key count rises, peaks, then declines while raw validity keeps climbing: `prooflite` peaks at round 6 (823 distinct keys) and falls to 686 by round 8 (§5.7); `stratlite` peaks at round 4 (694) and falls to 534 by round 7 (§5.6), the steeper drop of the two. Past the peak the policy trades breadth for reward — concentrating on a narrower band of easy, high-scoring programs — a mild mode-narrowing the anti-collapse guards of §5.5 bound but do not abolish. So "train longer" is not free: the generator has an optimal stop, and it lives in the distinct-key curve, not the rich-rate the runs actually early-stopped on. `prooflite`'s C6 was selected at its diversity peak; `stratlite`'s C7 was selected on rich-rate saturation and therefore sits past its own — its headline validity numbers are unaffected (validity is monotone), but its generator is less varied than an R4 checkpoint's would be. And narrowing is the milder half of what self-play does to the policy: §5.8 measures the sharper half — the reward's output shape persisting as an unconditional habit that overrides held-out specs, halving the plain-SFT checkpoint's problem-solving where spec and reward shape conflict — and then shows it is symmetric: retrain with correctness as the top rung and solving rises past plain SFT while free-form richness collapses instead. The imprinting is not a defect of one reward; it is what optimizing against any verifier does.

Two further edges of the fine-tune stay untested, and belong here rather than only inline in §5.7. First, cross-model generality: both arms fine-tune the same base model (`Qwen3-0.6B`), so the lift could in principle reflect that model's inductive biases rather than the recipe — nothing here varies the model. Second, the novelty control (§5.7) is measured only against the human cold-start corpus, the model's sole external data; it rules out memorizing those 174 examples, not reproduction from the far larger set of the model's own self-play programs, which was not separately checked. Neither undoes the results, but a skeptic reading the limits section should find both stated, not buried.

6.7 What smallness cannot buy

Smallness buys decidable *form*: this program halts, touches only these capabilities, emits at most this many bytes, is active enough to be a strategy. It does not buy *semantic correctness beyond the checked properties*, and on this corpus the gap is stark. All 132 survivors clear every mechanical gate — they compile, they provably halt, they respect the effect bound, they trade actively enough to pass the gate. Of those 132, 114 lose money on the very window they were evaluated against; only 18 are profitable on train (the `train_pnl` field over `class:"ok"` rows of the `reward` command in §6.1). The verifier certifies that a program is a well-formed, terminating, active strategy. It says nothing about whether that strategy is a *good* one, and by construction it must not: train PnL is deliberately excluded from the reward (§5.5), because rewarding it teaches curve-fitting and makes any held-out comparison circular.

This is the boundary of the entire approach, stated plainly. A purpose-sized language makes a chosen set of properties mechanical and complete; it makes no other property true. Edge, correctness of intent, robustness to a regime it never saw — these live outside the checked set and stay the job of selection, out-of-sample evaluation, and human judgment. The right reading of "132 survivors, 114 unprofitable" is not that the verifier failed but that it succeeded at exactly what it claims and nothing more: it separated well-formed strategies from noise, and left the question of which well-formed strategy is worth running entirely open. Where the properties a task needs fall outside any small decidable set — open-ended correctness, rich effects, behavior that only tests can pin down — a general-purpose language plus a test suite remains the better tool, and this kit makes no claim on that ground.

Appendix: Reproducibility

Every number in this paper is reproducible by a command in the repository, with one honest exception stated once and everywhere it applies: the *generation* of programs (by agents, a frozen API model, or a fine-tune) does not reproduce and is committed or recorded, not regenerated; the deterministic verifier plus the committed artifacts reproduce every figure below. All commands run from the repo root unless a `cd` is shown.

Kit structure, size, and caps — `bash scripts/caps.sh`

Number	Command
repo total 8,214 LOC (cap 25,000)	`bash scripts/caps.sh`
`diaglite` 252 LOC	`bash scripts/caps.sh`
`lexlite` 290 LOC	`bash scripts/caps.sh`
`parselite` 238 LOC	`bash scripts/caps.sh`
`fuellite` 180 LOC	`bash scripts/caps.sh`
`caplite` 521 LOC	`bash scripts/caps.sh`
`evmlite` 1,400 LOC	`bash scripts/caps.sh`
`modlite` 770 LOC	`bash scripts/caps.sh`
`prooflite` 1,955 LOC (cap 2,000)	`bash scripts/caps.sh`

Number	Command
`stratlite` 1,893 LOC	`bash scripts/caps.sh`
`backtestlite` 669 LOC	`bash scripts/caps.sh`
`stratlite` + `backtestlite` = 2,562 LOC	`bash scripts/caps.sh` (1,893 + 669)
kernel `diaglite`+`lexlite`+`parselite`+`fuellite` = 960 LOC	`bash scripts/caps.sh` (252+290+238+180)
seven kit crates = 3,651 LOC	`bash scripts/caps.sh` (sum)
emitters combined 2,170 LOC (built, unconsumed in repo)	`bash scripts/caps.sh` (1,400 + 770)
`experiment/src` 1,489 LOC (cap 1,500); harness 11 lines below ceiling	`bash scripts/caps.sh`
`experiment/train` 639 LOC (cap 800)	`bash scripts/caps.sh`
`experiment/proofbench/src` 354 LOC (cap 1,500) — the N=2 reward tool	`bash scripts/caps.sh`
`CLAUDE.md` 7,963 chars (cap 8,000)	`bash scripts/caps.sh`
per-crate cap 2,000, repo cap 25,000, `CLAUDE.md` cap 8,000	`scripts/caps.sh` (CRATE_CAP/REPO_CAP/CLAUDE_CAP)
zero external dependencies, all 11 crates	inspect each `crates/*/Cargo.toml` `[dependencies]` (workspace-internal entries only)
`wasm32-unknown-unknown` target green (exit 0)	`cargo check -target wasm32-unknown-unknown`

Test counts and pinned invariants — `cargo test -p <crate>`

Number	Command
`diaglite` 7 tests	`cargo test -p diaglite`
`lexlite` 7 tests	`cargo test -p lexlite`
`parselite` 4 tests	`cargo test -p parselite`
`fuellite` 4 tests	`cargo test -p fuellite`
`caplite` 10 tests	`cargo test -p caplite`
`evmlite` 17 tests	`cargo test -p evmlite`
`modlite` 8 tests + 1 doctest	`cargo test -p modlite`
`litelite` facade 1 test	`cargo test -p litelite`
58 unit tests + 1 doctest across the kit	sum of the eight kit crates above
`prooflite` 31 unit tests + 2 doctests	`cargo test -p prooflite`
`stratlite` 10 unit tests + 1 doctest (11 pass)	`cargo test -p stratlite`
`backtestlite` 6 unit tests + 1 doctest (7 pass)	`cargo test -p backtestlite`
cost model exact: `print 1;`=2, `let x = 1 + 2;`=4, `repeat 3 { print 0; }`=11	`cargo test -p prooflite` (`the_cost_model_is_exact`)
operator chain of 50 evaluates, 500 trips `E0102`, 5,000 parse-and-drop safe	`cargo test -p prooflite` (`operator_chains_count_toward_the_depth_cap`)
`caps()` fetched exactly once per run	`cargo test -p prooflite` (`the_validated_table_snapshot_drives_the_whole_run`)
`equity_hash` FNV-1a-64, determinism pinned	`cargo test -p backtestlite` (`determinism_is_exact_and_hashable`)

Number	Command
no-look-ahead prefix-invariance	`cargo test -p stratlite` (`no_lookahead_prefix_invariance`)

Source constants and signatures — `Read crates/... / experiment/...`

Number	Command
`DEFAULT_MAX_DEPTH = 96`	`crates/parselite/src/lib.rs:25`
`prooflite` default fuel 100,000 / output 64 KiB	`crates/prooflite/src/lib.rs:247-248` (`Limits::default`)
`stratlite` `fuel_per_bar = 25_000`	`crates/stratlite/src/lib.rs:137`
`verify() → Result<(Strategy, Report), Reject>`	`crates/backtestlite/src/lib.rs:199`
`enum Reject { Compile | Run | Gate }`	`crates/backtestlite/src/lib.rs:124`
gate default `min_trades = 4`, `min_bars_evaluated = 16`	`crates/backtestlite/src/lib.rs:89`
`stratlite::REFERENCE` generation prompt (a `const`)	`crates/stratlite/src/lib.rs:78`
M6 default model `Qwen/Qwen3-0.6B` (trainer config)	`experiment/train/train.py:55`
frozen model `claude-opus-4-8`	`experiment/src/api.rs:21`
A/B arm requires `ANTHROPIC_API_KEY` (permanently keyless)	`experiment/src/api.rs:69`

Construction cost — git history and `GENESIS.md`

Number	Command
genesis `a557fef` 2026-07-14 22:10:31	`git show -s -format='%h %ad' -date=format:'%Y-%m-%d %H:%M:%S' a557fef`
`prooflite` land `07c2877` 00:12:30, review `9ac20b3` 00:40:37 (28 min); M3 done `3d5918e` 02:25:10; `stratlite/backtestlite` land `42667a6` 02:59:24, review `1524eb3` 03:34:19 (35 min); M4 span 1 h 09 m; end-to-end 5 h 24 m	`git log -format='%h|%ad|%s' -date=format:'%Y-%m-%d %H:%M:%S' a557fef..1524eb3`
parents: `rustlite` 8.0K LOC/99 tests, `soliditylite` 7.6K/159, `bashlite` 3.1K/64 (external, not repo-reproducible)	`GENESIS.md`, "The lineage"
`bashlite` shipped with no parser depth guard	`GENESIS.md`, "What was ported at genesis"
`prooflite` review 16 raw / 13 confirmed / 2 crash-grade (M1); M2 16 confirmed; `stratlite`+`backtestlite` 14 confirmed (M4)	`GENESIS.md`, "Post-genesis lessons"
roadmap budgeted ~a week per language	`GENESIS.md`, "The three questions only reality can answer"

Number	Command
predecessor: 0.5B model 1.5% → 16.4% pass@1, 201-problem benchmark (tempo-x402, prior work, not recomputed here)	`GENESIS.md`, "The lineage"; `paper/OUTLINE.md`

Experiment run — corpus + verifier (all deterministic)

Number	Command
corpus 134 programs; trend 23 / breakout 23 / mean-reversion 22 / momentum 22 / stateful 22 / combo 22	`wc -l experiment/corpus/seed.jsonl`; `Counter` over the `style` field
744 candles; split train 0..446 / held-out 446..744; costs fee 2196 / slip 439 ticks (5 bps / 1 bps, train-only); drift 0.943x	`cd experiment && cargo run -q - data data/BTCUSDT-1h-2024-01.csv`
Reject histogram 0 compile / 0 run / 2 gate / 132 ok = 100% compile, 132/134 = 98.5% survivor	`cd experiment && cargo run -q - reward corpus/seed.jsonl data/BTCUSDT-1h-2024-01.csv` (tally `class`)
fuel over survivors min 16 / median 55 / max 186 vs 25,000 cap = 0.74% at max	`cd experiment && cargo run -q - reward corpus/seed.jsonl data/BTCUSDT-1h-2024-01.csv` (`fuel` over `class:"ok"`)
134 distinct `novelty_key` over 134 programs	`cd experiment && cargo run -q - reward corpus/seed.jsonl data/BTCUSDT-1h-2024-01.csv` (distinct `nkey`)
114 of 132 survivors negative `train_pnl`, 18 positive	`cd experiment && cargo run -q - reward corpus/seed.jsonl data/BTCUSDT-1h-2024-01.csv` (`train_pnl` over `class:"ok"`)
conditional eval: compile 100%, gate-clear train 98.5% vs held-out 97.0%, gap 1.5 points	`cd experiment && cargo run -q - eval corpus/seed.jsonl data/BTCUSDT-1h-2024-01.csv`
per-style held-out gate-clear: breakout 100% (23/23), combo 100% (22/22), momentum 100% (22/22), stateful 100% (22/22), trend 96% (22/23), mean-reversion 86% (19/22)	`cd experiment && cargo run -q - eval corpus/seed.jsonl data/BTCUSDT-1h-2024-01.csv`
reward ladder Compile 0 / Run 1/3 / Gate 2/3 / Ok 1; `value == ladder(class)`; anti-hacking guards; 23 tests pass (8 match `reward`)	`cd experiment && cargo test`
Python admission guards (Ok-rung only, `novelty_key` dedup, per-style cap) stdlib-only; 6 pass	`cd experiment/train && python3 test_select.py`

Fine-tune benchmark (M6) — reproducible scoring; model not reproducible

The MODEL does not reproduce (stochastic sampling; a fine-tune is not bit-identical across hardware). The SCORING does: the sample pools and candles are committed, and each row below

reproduces from a command. `target/` is git-ignored, so build the verifiers first on a clean clone: 'cd experiment && cargo build –release `(s5) and` cd experiment/proofbench && cargo build –release' (p6).

Number	Command
base compile / gate-clear 0.0% / 0.0% on all three windows	`cd experiment && ./target/release/s5 eval results/base.jsonl data/<window>.csv`
C7 BTC Jan 2024 (train window) 100.0% / 95.7%	`cd experiment && ./target/release/s5 eval results/c7.jsonl data/BTCUSDT-1h-2024-01.csv`
C7 BTC Jun 2024 (5-month embargo) 100.0% / 96.5%	`cd experiment && ./target/release/s5 eval results/c7.jsonl data/BTCUSDT-1h-2024-06.csv`
C7 ETH Jun 2024 (cross-asset) 100.0% / 96.1%	`cd experiment && ./target/release/s5 eval results/c7.jsonl data/ETHUSDT-1h-2024-06.csv`
train-minus-held-out gap 2.3 / 1.2 / 1.6 points (BTC Jan / BTC Jun / ETH Jun)	same `s5 eval` commands (TRAIN vs HELD-OUT line)
per-style held-out gate-clear near-uniform 88–100% on the cross-asset window	`cd experiment && ./target/release/s5 eval results/c7.jsonl data/ETHUSDT-1h-2024-06.csv`
training curve full-survivor cold-start → R0 39.0% → R7 98.2% (8 rounds); `distinct_nkeys` peaks R4 (694) then narrows to 534 (R7)	`experiment/results/train_curve.log`
pools 256 samples each (32/style × 8 styles); full benchmark output	`experiment/results/{base,c7}.jsonl`; `experiment/results/README.md`, `experiment/results/benchmark.txt`
C7 novelty 99.6% (250/251 gate-clearers ∉ corpus)	`cd experiment && ./target/release/s5 reward results/c7.jsonl data/BTCUSDT-1h-2024-01.csv` (compare `nkey` field vs `s5 reward corpus/seed.jsonl ...`)
N=2 prooflite: base RICH 3.5% (9/256), C6 RICH 94.5%	`cd experiment/proofbench && ./target/release/p6 eval results/base.jsonl` then `... eval results/c6.jsonl` (one pool per run)
N=2 prooflite: novel rich ~100% (∉ 174-program corpus); C6 216 distinct rich	`cd experiment/proofbench && ./target/release/p6 novelty results/c6.jsonl corpus/seed.jsonl`
N=2 prooflite training curve rich-rate R0 43.8% → R8 96.2% (9 rounds); `distinct_nkeys` peaks R6 (823) then narrows to 686 (R8)	`experiment/proofbench/results/train_curve.log`
N=2 prooflite pools 256 each; full output	`experiment/proofbench/results/{base,c6,c7,c8}.jsonl`; `experiment/proofbench/results/README.md`, `.../benchmark.txt`

PENDING slot — instrument shipped, run not performed (permanently keyless)

Slot	Command
A/B arm: generate	`cd experiment && cargo run - submit <n> batch.json` then `cargo run - poll <batch_id> raw.jsonl` (needs `ANTHROPIC_API_KEY`)
A/B arm: score the two arms (pure, no network)	`cd experiment && cargo run -q - score raw.jsonl data/BTCUSDT-1h-2024-01.csv`