

Compact recurrence encodings for code generation: A four-model study of program length, correctness and token usage

Sharjeel Abbas

ORCID: 0009-0003-2144-900X

2 October 2026

Abstract

Compact program encodings can reduce emitted code while changing the difficulty and token expenditure of generating a correct solution. We define and evaluate an explicit typed recurrence notation, a shorthand with the same checked semantics, and templates for common recurrence shapes. Two Rust controls distinguish general code generation from access to shared components. Four hosted model configurations each complete 23 tasks, five arms and three repetitions, with at most two feedback-conditioned repairs. The tasks comprise adapted HumanEval+ and MBPP+ subsets and 12 study-authored problems. Canonical shorthand references use 42.35% fewer auxiliary tokens than explicit Core references. Generated shorthand initial programs are 25.5% to 40.8% shorter across the models, including unsuccessful programs. Final Core versus Shorthand solved counts out of 69 are 69 versus 56 for Gemini, 58 versus 44 for Sonnet, 65 versus 47 for Luna and 69 versus 67 for Sol. Task-level bootstrap intervals favor Core in the first three configurations; Sol’s initial comparison is close and uncertain. Gemini, Sonnet and Luna use 1.75, 1.32 and 2.60 times as much reported output per solved repetition in Shorthand as in Core. Sol’s complete cost ratio is unknown because three failed Core calls lack usage. We retain all 1,962 raw calls, including 32 failed transports, and audit both Codex models for forbidden tool attempts. The findings separate representational compression from successful generation under these hosted interfaces. They do not isolate familiarity, reasoning effort or a causal effect of notation alone.

1 Introduction

Code-generating models consume and emit token sequences. A programming notation can remove repeated types, abbreviate common operations or replace an explicit table definition with a compact template. Such changes can shorten equivalent programs. The relevant generation budget also includes unsuccessful answers, repairs, prompt history and any reported thinking tokens.

This study examines Intent, a checked language for recurrences over byte-string inputs. It has an explicit typed Core notation, a shorter notation with inferred declarations and complementary cases, and a Shape notation for common one- and two-dimensional tables. The compiler checks programs and emits Rust. The current work evaluates those text interfaces and their schemas. Training models to use them and designing a later language with its own execution system are separate future steps.

We compare five arms under a frozen task protocol. Core and Shorthand share recurrence operations, checks, components and canonical semantics. This is the closest comparison of surface notation in the experiment. Shape further abbreviates table structure. Standalone Rust and Rust with a shared library provide controls with a more expressive programming model. Their comparison with Intent therefore includes differences in expressiveness and static restrictions.

Four model configurations complete the same 345 initial task repetitions and a bounded repair protocol. The study asks whether shorter program text accompanies lower reported output, how repairs change the comparison, and how the findings vary across models and task strata. We report each model separately. No model-specific training occurs, model identities and effort settings remain fixed, and no extra model enters the experiment.

The original six-task pilot manuscript remains an independent preserved artifact. Its generation results are not included in this study’s statistics. The present batch adds an explicit Core arm, shared generic components,

audited input adaptation and a larger frozen corpus. These changes prevent a direct before-and-after attribution to a single implementation change.

2 Related work

SimPy changes Python’s grammar while preserving abstract syntax trees. Sun et al. report token reductions of 13.5% for CodeLlama and 10.4% for GPT-4 on their code-related tasks [1]. Token Sugar mines 799 reversible shorthand patterns and trains models on transformed source; its reported code-token reductions reach 15.1% and generation-token savings reach 11.2% [2]. ShortCoder combines validated Python simplification pairs with fine-tuning and reports an 18.1% source-token reduction [3]. Those training procedures and efficiency measures differ from prompting existing hosted models to use a restricted recurrence language.

CFGzip compresses a grammar engine’s token search space. Its October 2026 revision reports up to a 75-fold latency reduction during batched inference [4]. It concerns grammar-engine latency, rather than shorter program output or reduced model thinking. These approaches may complement a compact language, but the present experiment does not test their combination.

HumanEval measures functional correctness of generated code [6]. MBPP provides programming problems with natural-language descriptions and tests [5]. EvalPlus supplies stronger test corpora [7], and MultiPL-E studies test-driven evaluation across languages [8]. We use finite behavioral tests on adapted subsets selected for representability. The results are not full-benchmark leaderboard scores.

Coconut trains a model to feed continuous hidden-state representations back as thoughts [9]. Our encodings change requested program text. Provider-reported thinking usage does not reveal the internal representations or reasoning process, and this study makes no claim about latent-space reasoning. The cited literature also does not support a claim that this is the first study to consider reasoning overhead.

Halide separates an algorithm from the schedule that organizes its execution [11]. Intent likewise separates recurrence equations from validated storage and arithmetic plans, but this study evaluates generated program text and functional success. It does not compare compiler performance with Halide or establish a runtime speedup from compact notation.

Chain-of-thought prompting studies explicit intermediate reasoning [12]. Test-time scaling studies how additional inference computation and its allocation affect success [13], while s1 uses budget forcing to control the length of a trained model’s reasoning [14]. These motivate measuring the complete generation budget rather than the final code alone. Our hosted calls expose usage categories, not a common compute counter, and this experiment neither changes thinking budgets within a model nor applies budget forcing.

Self-Refine uses model-generated feedback to revise an answer [15]. CodeT uses generated tests and execution agreement to select code [16]. Process supervision instead trains a verifier on intermediate reasoning steps [17]. Our repairs use compiler diagnostics or a failing test counterexample, with a fixed two-repair limit; we do not train a verifier or use model-generated tests as the correctness oracle. Work on chain-of-thought faithfulness further cautions against treating a reported reasoning trace as a complete account of internal computation [18]. A token count is a usage measurement, not an inspection of the model’s reasoning process.

3 Language and encoding schemas

3.1 Checked recurrence semantics

Intent functions accept byte strings and return an unsigned 64-bit Index. A recurrence defines a table with one or two axes and returns a designated cell. Axis bounds are inclusive and use input lengths or integer constants. Each cell must have a definition. Cases are unordered, and overlapping cases must agree. The final Shorthand `else` denotes the complement of all earlier cases; Core writes that complement as an explicit guard.

Guards compare affine expressions of axes, input lengths and constants, with Boolean combinations. They cannot read bytes or table cells, use division or modulo, or multiply variable expressions. Cell values can compare bytes and use arithmetic, `min` and `max`. Core and Shorthand convert a Boolean to an Index with `bit`. Byte comparisons require byte operands. Arithmetic checks overflow, negative subtraction and division by zero.

Table dependencies use fixed offsets and must precede the current cell in row order. An access such as `T[i-1,j]` is permitted; an access to a dynamically chosen column is not. The checker proves byte indices safe for all input lengths, including empty strings, even when a task contract excludes empty inputs. Both operands

of Boolean conjunction and disjunction evaluate, so those operators cannot protect an unsafe byte access. These rules define a restricted programming model, rather than a general replacement for Rust.

The compiler checks coverage, overlapping cases, types, safe indexing and dependency order before it emits native code. These obligations do not prove the generated program satisfies the natural-language task. A statically accepted program can still produce wrong results, and a finite test pass is not a universal correctness proof. Checked arithmetic may also fail on sufficiently large inputs beyond the evaluation corpus.

3.2 Core and Shorthand

Core declares typed parameters, a return type, an `eq` recurrence, explicit domains, `len` expressions and explicit case guards. Shorthand omits types, infers the table bounds from the result cell, uses `#a` for the input length and supplies a complementary `else`. Both arms receive the same component catalog and semantic restrictions.

The frozen guides contain equivalent generic edit-distance examples. The Core form is

```
fn distance(a: Bytes, b: Bytes) -> Index = eq T[len(a),len(b)] {
  domain i in 0..=len(a), j in 0..=len(b)
  T[i,0] = i
  T[0,j] = j
  T[i,j] when j!=0 && i!=0 =
    min(T[i-1,j]+1,T[i,j-1]+1,T[i-1,j-1]+bit(a[i-1]!=b[j-1]))
}
```

The Shorthand form is

```
fn distance(a,b) = T[#a,#b] {
  T[i,0] = i
  T[0,j] = j
  else T[i,j] =
    min(T[i-1,j]+1,T[i,j-1]+1,T[i-1,j-1]+bit(a[i-1]!=b[j-1]))
}
```

The task references pass every associated evaluation case. Their canonical semantic digests agree between Core and Shorthand. This agreement checks the reference translation, not the correctness of all model-generated programs. The Core arm also enforces its explicit declaration format, which creates an arm-adherence requirement in addition to ordinary compilation.

In this example, $T[i, j]$ is the minimum number of edits between the first i bytes of a and the first j bytes of b . The result cell $T[\#a, \#b]$ selects the complete inputs. The two boundaries describe deletion of an entire prefix and insertion into an empty prefix. At the corner, both equations give zero and therefore agree. For an interior cell, the three candidates are deletion, insertion and substitution or a free byte match. The expression $\text{bit}(a[i-1] \neq b[j-1])$ is one for a mismatch and zero for a match. On `kitten` and `sitting`, the result is three.

Shorthand text	Explicit meaning recovered by the compiler
<code>a, b</code>	Both parameters have type <code>Bytes</code> ; the function returns <code>Index</code> .
<code>#a, #b</code>	Lengths <code>len(a)</code> and <code>len(b)</code> .
<code>T[#a, #b]</code>	Result cell and inclusive axis bounds $0 \leq i \leq \#a$, $0 \leq j \leq \#b$.
Omitted <code>eq</code>	The body is a recurrence equation.
Final <code>else</code>	The complement of all preceding cases, here $i \neq 0$ and $j \neq 0$.

Table 1: Deterministic expansion rules for the edit-distance Shorthand example. Removing declarations does not remove their meaning or the checks applied to the expanded program.

The default parameter and result types are fixed rules, rather than task-dependent guesses. Inferred bounds must be supported input lengths or constants. The final `else` cannot mean an arbitrary exception or an imperative branch selected by case order. Its complement must be expressible in the supported guard language.

3.3 Shape templates and reusable components

Shape defines a line or grid and uses aliases such as `prev`, `up`, `left` and `diag`. For example,

```
distance = grid col i row j min(up+1,left+1,diag+diff)
```

expands to the corresponding checked table. Shape has its own Boolean-to-number conventions, unlike Core and Shorthand. These parser and typing details remain part of the evaluated interface. A unary Shape may omit the redundant second input parameter; its adapter passes the first input and ignores the duplicate copy.

For the grid example, `col i` sets $T[i, 0] = i$ and `row j` sets $T[0, j] = j$. The aliases `up`, `left` and `diag` substitute $T[i - 1, j]$, $T[i, j - 1]$ and $T[i - 1, j - 1]$. In a grid, `diff` substitutes the mismatch of $a[i - 1]$ and $b[j - 1]$. The grid supplies both byte parameters, both length bounds and the final result cell. Figure 1 shows the corresponding dependencies. Omitting the two edge declarations would select zero-valued boundaries and would change this edit-distance algorithm.

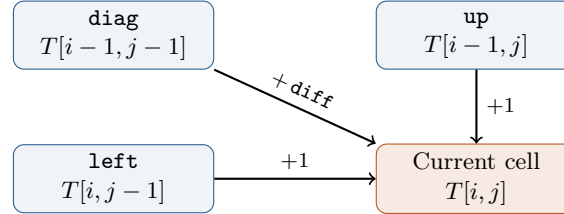


Figure 1: The edit-distance grid chooses the minimum of three candidates. Shorthand spells out these reads; Shape names them with aliases. Boundary equations handle empty prefixes before an interior expression accesses bytes or earlier cells.

A line template supplies a one-dimensional table and the result $T[\#a]$. Consider counting maximal runs of equal bytes. The following Shorthand and Shape programs return three on `aaabbc` and zero on an empty input:

```

fn runs(a,b) = T[#a] {
  T[0] = 0
  T[1] = 1
  else T[i] = T[i-1]+bit(a[i-1]!=a[i-2])
}

runs = line at 0 = 0 at 1 = 1 prev+diff

```

Here `prev` means $T[i - 1]$. In a line, `diff` compares the current byte $a[i - 1]$ with the preceding byte $a[i - 2]$. The first byte creates one run; every later change starts another. The explicit cells at zero and one prevent the general expression from reading nonexistent bytes. The unary Shape declares only a ; the experiment adapter ignores the redundant b copy. Thus the two forms have the same task outputs, but the unary Shape and the two-parameter Shorthand do not have identical function signatures. The meaning of `diff` depends on whether the template is a grid or a line.

Shape converts a comparison directly used in arithmetic to zero or one. It does not accept `bit(...)` in the evaluated parser. For example, `prev*(x==first)` is numeric, whereas `prev&&(x==first)` mixes numeric and Boolean values and fails typing. A comparison used as a complete numeric cell value, or inside `min` or `max`, needs a numeric wrapper such as `0+(x==first)`. These conventions are part of the interface and can make a short expression harder to author correctly.

Nine generic components are available at their default constants: edit distance, longest common subsequence, subsequence membership, distinct-subsequence counting, common prefix, adjacent repeats, runs, byte counting and lattice paths. Intent allows a component call only as the complete function body, with declared input parameters as arguments. No task-specific helper is added. The Rust library arm receives these same components as `seed::` functions. Standalone Rust uses the standard library and ordinary helpers written in the answer, with no external crates.

3.4 Authoring and checking practice

The intended workflow starts with an input/output contract and a recurrence whose boundary values are explicit. A matching generic component can replace that recurrence only when its byte semantics and return contract fit. Otherwise the author writes Core or Shorthand, checks coverage, overlap and dependencies, and inspects the normalized Core expansion. A Shape is suitable when its line or grid bounds, edge conventions and aliases describe the same computation. Unsupported constructs require a different implementation; the compiler does not invent a meaning for them.

The project commands `check` and `explain` expose the checked equations and execution plan. Component calls require `lock` to record the components and their digests. Behavioral `test` cases should include empty inputs, one-byte inputs, equal and unequal lengths, and independent expected values appropriate to the task. `build` emits and builds Rust; `run` requires the current built artifact and rejects stale source or configuration. Command-line strings use UTF-8 bytes, whereas this experiment’s adapter uses ranked codepoints. These are distinct input contracts.

Static acceptance establishes the supported language obligations, not the natural-language task. A rewrite should retain the same normalized semantics where signatures agree and pass independent behavioral checks. Source length, complete generation usage and success after repairs must then be assessed separately. The examples in this section illustrate the frozen grammar and have been checked locally; they do not add tasks or calls to the experiment.

4 Experimental protocol

4.1 Tasks and adaptation

The frozen corpus has 23 tasks. Six HumanEval+ tasks from release v0.1.10 contribute 5,900 base and expanded cases. Five MBPP+ tasks from release v0.2.0 contribute 527 cases. Twelve study-authored synthetic tasks contribute 5,868 cases. Appendix A lists every task and case count. All arms use the same expected outputs.

The selection requires one or two string inputs and an unsigned-integer or predicate output that fits the restricted recurrence model. Selection also requires a checked reference program. This favors representable problems and limits external validity. The MBPP adapter audit retains the selected corpus under its adapted contracts; we do not treat its scores as the official unmodified MBPP metric.

For each case, the adapter ranks the union of Unicode codepoints in sorted order and maps them injectively to bytes 1 through 127. It preserves equality, codepoint order and length. The function ABI is `pub fn ENTRY(a: &[u8], b: &[u8]) -> u64`. Unary tasks duplicate the first input as the second slice. Predicate outputs must be exactly zero or one. The supported alphabet is limited to 127 distinct codepoints per case; category-dependent, literal-character and structured-output tasks are outside the operational scope.

The synthetic tasks test pair comparisons, local patterns, fixed-position comparisons, runs, substring recurrences and weighted alignment. They are study-authored. Their inclusion does not establish that the models had never seen equivalent tasks or algorithms. A sensitivity analysis removes four especially simple MBPP tasks, identifiers 79, 238, 242 and 741, and retains MBPP/247.

4.2 Models, prompting and bounded repairs

The model identifiers are `gemini-3.8-flash-low` and `claude-sonnet-4-6` through Antigravity, `gpt-5.6-luna` with high effort through Codex CLI, and `gpt-6.1-sol` with medium effort through Codex CLI. The short labels in the tables identify these configurations. Sol is the fourth and final approved model. Its tasks, arm guides, compiler, ABI, initial prompts, repetitions and repair policy match Luna’s. Medium Sol versus high Luna confounds model and effort.

Each model completes 23 tasks times five arms times three repetitions, for 345 usable initial answers. A wrong usable answer receives at most two repairs. The repair prompt includes the prior prompt history, the extracted answer and compiler feedback or a failing test counterexample. Hidden tests and reference answers do not appear in the initial prompt. The references reside in the local evaluator, separate from the generation invocation.

All 345 exact initial prompts agree across the four models. Generic syntax examples and component catalogs differ by arm as required by their interfaces. Under the auxiliary tokenizer, mean initial prompt lengths are 237.8 tokens for Rust, 508.8 for Rust with a library, 1,091.8 for Core, 1,021.8 for Shorthand and 1,441.8 for Shape. Thus the closest notation comparison also changes guide text and removes about 70 visible prompt tokens. Reported provider input can include additional client context and follows provider-specific cache conventions.

Job ordering uses seed 61004. Sampling seeds and temperature are unavailable through these invocations. Repetitions are repeated trials under recorded settings, without a guarantee of independent draws from a known sampling distribution. The initial three-model queue shares four workers. Recovery continuations use one worker until the user’s final Sonnet continuation increases concurrency to four. Sol uses one worker through-

out. We report concurrency and interruptions as operational provenance rather than comparable throughput measurements.

Tools are forbidden. Codex runs ignore user configuration, use ephemeral sessions, a read-only sandbox and a fresh empty working directory, with the task prompt on standard input. Both started and completed non-answer tool events disqualify an answer. The complete audit examines 537 Luna traces and 409 Sol traces and finds zero such attempts. A read-only sandbox does not itself prove that all filesystem reads were inaccessible; the recorded absence of tool attempts is the relevant evidence.

4.3 Execution, transport and preservation

The frozen compiler source commit is 6697ca362add283b63321f48aed09ea8685ddda2. Rust 1.94.1 builds the native drivers with edition 2024 and optimization enabled. Compilation and runtime limits are 120 seconds each. A correct answer must produce exactly the required number of outputs, with every value matching the expected result. Panics, arithmetic errors, missing rows and malformed outputs fail.

Preflight validates the selected Core and Shorthand references against the complete case corpus, checks generic guide examples and exercises both Rust controls. Five cases per task cross-check reference interpretation against native output. This is a spot-check of implementation agreement. Fourteen frozen preparation tests and eight analysis tests pass; Sol also passes its saved runner and prompt-parity checks. No compiler, task, prompt or guide changes occur during generation.

Calls retain raw prompts, replies, errors and usage metadata. Pre-generation retryable 503 or 429 failures may receive up to two retries, while other failed calls remain infrastructure attempts for a later continuation. A usable wrong program advances the bounded algorithmic repair count; a failed transport does not consume that count. Client print timeouts are 300 seconds and transport limits are 330 seconds in this batch. Host sleep can distort wall-clock comparisons and is recorded separately.

Every completed answer is saved immediately. Checkpoints preserve exact result bytes and raw-call inventories. Continuations copy prior results and calls, retain failed attempts and start the first pending round. Their latest result files are cumulative. We never add predecessor totals to them. Frozen manifests and the original pilot manuscript remain unchanged. The final audit checks every registered segment against its saved hashes and reconciles raw call files with cumulative totals.

4.4 Metrics and uncertainty

For each arm, $N = 69$ task repetitions. Let S_r count repetitions solved by round r , where zero denotes the initial answer and two denotes the final permitted repair. Success means at least one answer through that round passes all cases. Repairs depend on feedback, so the resulting success rate is not standard independent-sample Pass@3.

For a call c , I_c is reported input, O_c is reported output including thinking, and H_c is reported thinking. We use the provider’s thinking field as the reported reasoning-token category; it is not a provider-independent measurement of internal computation. We retain every call, including failed transports. We report total output and

$$C_a^{\text{out}} = \frac{\sum_{c \in a} O_c}{S_2}, \quad C_a^{\text{raw}} = \frac{\sum_{c \in a} (I_c + O_c)}{S_2}.$$

These ratios allocate a batch’s tokens across its solved repetitions. They are not dollar prices, compute measurements or estimates for unlimited retry. Thinking is already included in output and is not added again. Cache counts stay separate; we do not subtract them without a provider-specific rule. Reported zero thinking for Sonnet does not demonstrate no internal reasoning.

If usage is missing, the complete total and its cost ratio remain unknown. The sum of available nonnegative counts is a lower bound on a total, but a ratio of two available sums is not automatically a lower bound on their true ratio. In particular, three failed Sol Core calls have unknown usage, so its complete Shorthand/Core cost comparison cannot be determined.

Program length uses `o200k_base` with `tiktoken` 0.13.0 [10]. It is a common auxiliary tokenizer, not an assertion about each model’s tokenizer. We measure extracted program text independently of reported output and thinking. The initial length statistic includes invalid and incorrect programs. Appendix D also sums the extracted code across initial answers and all usable repairs. Canonical reference lengths provide a separate measurement of equivalent checked solutions.

For Core versus Shorthand, we match tasks and repetitions. We bootstrap 23 task blocks, each with its three repetitions, for 10,000 draws using seed 61005. The percentile interval estimates the difference in accuracy for this selected sample under task resampling. Exact task sign-flip tests assume exchangeable arm labels within task blocks. Cell-level McNemar values are descriptive because repetitions can correlate. All tests and sensitivity analyses are exploratory, without multiplicity adjustment or a preregistered population claim.

5 Results

5.1 Completion and correctness

All four models exhaust the planned protocol. The audit reconciles 1,962 raw calls, 1,380 usable initial answers, 550 usable repairs and 32 failed transports. There are no scoring failures. Gemini has eight failed transports, Sonnet 21, Luna zero and Sol three. One Sonnet failure and all three Sol failures have unknown input and output usage. Table 2 reports results separately for each model and arm. Rust with a library solves all 69

Model	Arm	Initial	Repair 1	Final	Calls	Failed
Gemini low	Rust	67	69	69	72	1
	Rust + library	66	69	69	75	3
	Core	47	62	69	99	1
	Shorthand	39	51	56	119	2
	Shape	43	53	55	112	1
Sonnet	Rust	69	69	69	72	3
	Rust + library	66	69	69	73	1
	Core	42	56	58	113	4
	Shorthand	25	37	44	155	10
	Shape	35	49	54	126	3
Luna high	Rust	51	68	68	88	0
	Rust + library	64	69	69	74	0
	Core	47	62	65	98	0
	Shorthand	31	41	47	135	0
	Shape	28	37	46	142	0
Sol medium	Rust	64	69	69	74	0
	Rust + library	69	69	69	69	0
	Core	55	67	69	88	3
	Shorthand	56	66	67	85	0
	Shape	52	62	67	93	0

Table 2: Solved repetitions out of 69 per model and arm. Repair columns are cumulative. Calls include all infrastructure attempts. Failed denotes failed transports, rather than incorrect programs.

repetitions for every model after repairs. Standalone Rust also solves all 69 for Gemini, Sonnet and Sol; Luna solves 68. Among recurrence encodings, Sol ends at 69 for Core and 67 each for Shorthand and Shape. Gemini reaches 69 with Core but only 56 and 55 with the more compact interfaces. Sonnet and Luna also end with fewer solved repetitions in Shorthand than Core. These counts show configuration-dependent behavior, without a causal ranking of model capability.

5.2 Matched Core and Shorthand comparison

Table 3 and Figure 2 give matched differences. Gemini’s initial difference is uncertain under the task bootstrap, but its final interval favors Core. Sonnet and Luna’s initial and final intervals favor Core in this sample. Sol solves one more initial repetition in Shorthand, with an interval that includes zero, and two fewer after repairs. These results do not establish Shorthand superiority or equivalence for Sol. The tasks are few and selected for representability. Intervals and exploratory probabilities describe this design and its resampling assumptions. They do not remove model training exposure, account variation or provider changes. Task-level contrasts are preferable to a pooled cross-model significance claim.

Model	Stage	Difference	95% interval	Core/Short only	p
Gemini low	Initial	-11.6	[-27.5, +5.8]	15/7	0.2499
	Final	-18.8	[-33.3, -7.2]	13/0	0.0156
Sonnet	Initial	-24.6	[-39.1, -10.1]	21/4	0.0059
	Final	-20.3	[-37.7, -4.3]	18/4	0.0459
Luna high	Initial	-23.2	[-34.8, -13.0]	17/1	0.0005
	Final	-26.1	[-39.1, -14.5]	19/1	0.0010
Sol medium	Initial	+1.4	[-11.6, +14.5]	5/6	1.0000
	Final	-2.9	[-7.2, +0.0]	2/0	0.5000

Table 3: Shorthand minus Core accuracy in percentage points, matched by task and repetition. Bootstrap intervals resample 23 tasks with three repetitions together. The last column is the exploratory exact task sign-flip probability. No adjustment for multiple comparisons is applied.

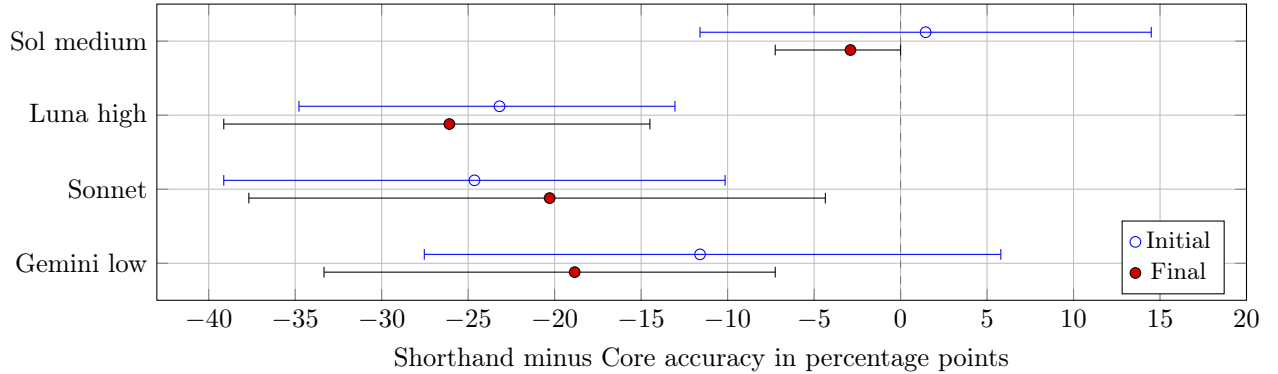


Figure 2: Matched accuracy differences with task-bootstrap intervals. Points to the left of zero favor explicit Core notation. The model configurations and effort settings differ across models.

5.3 Program length and reported token expenditure

The matched canonical references occupy 3,410 Core tokens and 1,966 Shorthand tokens, a 42.35% reduction. Within the synthetic, HumanEval+ and MBPP+ strata, the reductions are 45.12%, 35.21% and 52.87%. Every paired reference has the same canonical digest and passes its cases. The references are withheld from initial model prompts.

Generated initial Shorthand programs are 33.53%, 40.76%, 34.79% and 25.52% shorter than Core for Gemini, Sonnet, Luna and Sol. Those reductions include invalid and wrong answers. Table 4 separates program text from reported output. Gemini and Luna’s initial Shorthand output is about 1.21 times Core despite shorter

Model	Arm	Initial code mean	Initial O	All O	All H	O/S
Gemini low	Rust	99.5	38,333	39,063	30,189	566.1
	Rust + library	90.6	29,932	30,148	21,608	436.9
	Core	117.9	220,993	382,549	320,548	5544.2
	Shorthand	78.4	266,304	542,545	415,650	9688.3
	Shape	34.6	240,420	500,116	366,631	9093.0
Sonnet	Rust	109.6	15,658 [†]	15,658 [†]	0 [†]	226.9 [†]
	Rust + library	109.6	29,801	32,017	0	464.0
	Core	101.6	252,610	443,591	0	7648.1
	Shorthand	60.2	203,907	443,528	0	10080.2
	Shape	30.1	208,920	392,100	0	7261.1
Luna high	Rust	106.2	22,032	26,747	16,888	393.3
	Rust + library	89.2	18,591	19,596	12,693	284.0
	Core	130.4	127,441	210,249	194,233	3234.6
	Shorthand	85.0	154,743	395,803	381,026	8421.3
	Shape	44.1	136,369	251,206	242,972	5461.0
Sol medium	Rust	131.4	12,887	13,401	3,441	194.2
	Rust + library	77.3	7,991	7,991	2,324	115.8
	Core	107.2	20,619 [†]	28,621 [†]	17,636 [†]	414.8 [†]
	Shorthand	79.9	30,800	39,977	31,789	596.7
	Shape	36.0	17,709	30,645	25,508	457.4

Table 4: Initial code mean is the mean extracted program length for initial answers in auxiliary o200k tokens. It excludes reasoning and surrounding response prose and includes incorrect and invalid programs. O is reported output including thinking, H is reported thinking, and S is final solved repetitions. A dagger marks available partial totals or lower bounds because usage is unknown for one Sonnet Rust call and three Sol Core calls. H is already included in O . Sonnet’s reported zero thinking does not establish an absence of internal reasoning.

code. Their complete Shorthand output is 1.42 and 1.88 times Core, respectively. Once divided by final solved repetitions, those ratios rise to 1.75 and 2.60. Their raw input-plus-output ratios per solved repetition are 1.61 and 1.97. These are within-model batch ratios, rather than portable cost claims.

Sonnet shows a different initial pattern. Shorthand’s initial output is 0.81 times Core, and total output is almost identical, 443,528 versus 443,591 tokens. It solves fewer repetitions, so output per solved repetition is 1.32 times Core and raw input-plus-output per solved repetition is 1.57 times Core. A claim that compact notation always increases output would contradict this initial result.

Sol reports only 28,621 available Core output tokens against 39,977 known Shorthand output tokens. The unknown usage of its failed Core calls prevents a complete cost ratio. Its available sums can be reported, but they cannot establish that Shorthand had a larger actual total cost. Appendix B records input, cache and unknown counts.

5.4 Code-only tokens across initial answers and repairs

Appendix D adds code-only maps for every model, task and scheme. The initial maps compare three extracted programs per cell. The cumulative maps add their repair programs and therefore measure both program length and the amount of repair code produced. Reasoning and surrounding prose are excluded from both. We count

extracted text directly; subtracting thinking from reported output would leave other response content and would not isolate code.

Gemini low emits 8,135 initial Core code tokens and 5,407 initial Shorthand code tokens across 69 programs per arm. Including usable repairs, those totals become 13,681 and 10,072, respectively. Sonnet emits 7,009 initial Core code tokens and 4,152 initial Shorthand code tokens across 69 programs per arm. Including usable repairs, those totals become 11,023 and 8,832, respectively. Luna high emits 8,998 initial Core code tokens and 5,868 initial Shorthand code tokens across 69 programs per arm. Including usable repairs, those totals become 15,122 and 13,732, respectively. Sol medium emits 7,399 initial Core code tokens and 5,511 initial Shorthand code tokens across 69 programs per arm. Including usable repairs, those totals become 10,485 and 7,696, respectively. These auxiliary code totals can be complete even when provider usage is missing for failed calls. Shorter extracted code alone does not establish lower total generation usage or higher correctness.

5.5 Reported thinking and output by scheme

Figure 3 displays complete-protocol usage by model and scheme. Output includes its reported thinking portion, so the two bars are overlapping categories and must not be added. Larger totals can reflect more expensive individual calls, more repairs, infrastructure attempts, or their combination. Every scheme begins with the same 69 usable initial repetitions, but the subsequent call counts differ.

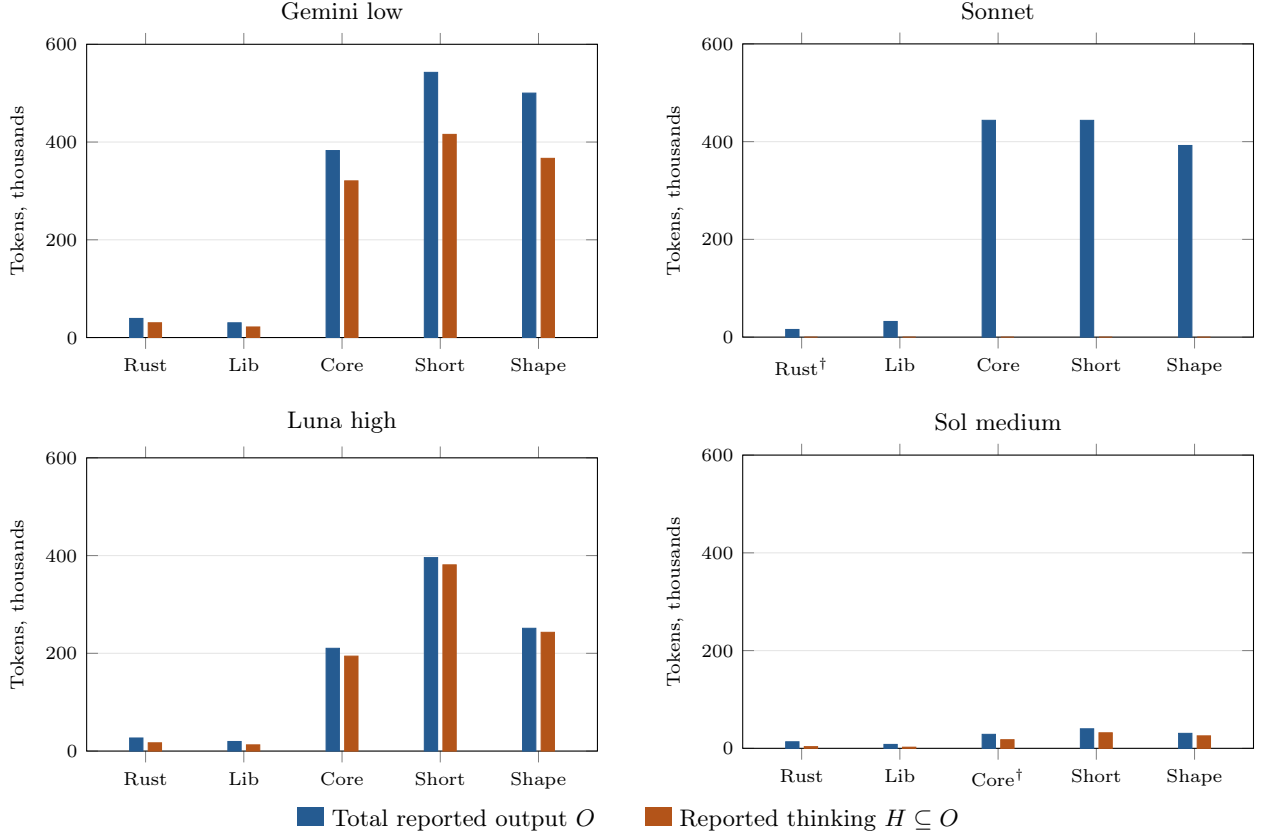


Figure 3: Usage by model and scheme across all tasks, repetitions, repairs and infrastructure calls. All four panels share the same vertical scale. Lib denotes Rust with the shared library. A dagger marks an available lower bound with unknown calls. Sonnet reports zero thinking on known calls; its internal reasoning is not separately measurable here. The panels describe each recorded configuration and do not rank compute efficiency across providers or effort settings.

Gemini reports 1,154,626 thinking tokens out of 1,494,421 output tokens, a 77.3% share. Luna reports 847,812 out of 903,601, a 93.8% share. Sol’s available counts are 80,698 thinking and 120,635 output tokens, with three unknown calls; its complete thinking share is unknown. Sonnet reports zero thinking on known calls

and 1,326,894 available output tokens, with one unknown call. Its reported zero category cannot identify the actual allocation between reasoning and answer generation.

Appendix C gives exact task-by-scheme counts for each model. Each heatmap cell sums three repetitions and any resulting repair and infrastructure calls. These are protocol totals, not per-call averages or cost per correct answer. Task labels retain the adapted HumanEval+, adapted MBPP+ and study-authored strata. High token usage does not imply that the task was solved; accuracy remains a separate outcome in Table 2.

5.6 Strata, sensitivity and rejection

Table 5 keeps the adapted benchmarks and synthetic tasks separate. Sol’s initial Shorthand advantage in the combined corpus includes MBPP+, where it solves 14 versus nine for Core. On synthetic tasks it solves 30 versus 34. The combined one-cell difference therefore does not indicate a uniform effect across task types. After

Model	Stratum	N	Rust	Library	Core	Short	Shape
Gemini low	HumanEval+	18	18/18	18/18	13/18	10/14	9/12
	MBPP+	15	15/15	15/15	12/15	12/15	13/13
	Synthetic	36	34/36	33/36	22/36	17/27	21/30
Sonnet	HumanEval+	18	18/18	18/18	7/14	6/11	11/13
	MBPP+	15	15/15	14/15	13/14	7/15	11/13
	Synthetic	36	36/36	34/36	22/30	12/18	13/28
Luna high	HumanEval+	18	17/18	18/18	10/17	6/12	8/14
	MBPP+	15	2/14	11/15	14/14	11/13	12/13
	Synthetic	36	32/36	35/36	23/34	14/22	8/19
Sol medium	HumanEval+	18	18/18	18/18	12/18	12/17	11/16
	MBPP+	15	15/15	15/15	9/15	14/15	15/15
	Synthetic	36	31/36	36/36	34/36	30/35	26/36

Table 5: Initial/final solved counts by model and task stratum. Each denominator includes three repetitions per task. The synthetic tasks are study-authored. HumanEval+ and MBPP+ are adapted subsets.

excluding four simple MBPP tasks, Core still exceeds Shorthand in final counts for every model. The intervals for Gemini, Sonnet and Luna remain below zero; Sol’s interval reaches zero. Table 6 reports this sensitivity analysis. It does not convert the remaining selected tasks into a random sample of general programming problems. Initial

Model	Rust	Library	Core	Short	Shape	Final difference interval
Gemini low	57	57	57	44	43	[-38.6, -8.8]
Sonnet	57	57	46	32	42	[-45.6, -5.3]
Luna high	57	57	53	35	34	[-45.6, -17.5]
Sol medium	57	57	57	55	55	[-8.8, +0.0]

Table 6: Sensitivity analysis excluding MBPP identifiers 79, 238, 242 and 741. Each arm has 57 repetitions across 19 tasks. The last column is the 95% task-bootstrap interval for Shorthand minus Core in percentage points. MBPP/247 remains in the analysis.

invalidity is common in recurrence arms. Sonnet has 25 invalid Core answers, 41 invalid Shorthand answers and 30 invalid Shape answers out of 69 each. Sol has 14, 13 and 15. Luna also has valid but wrong recurrence answers, including 13 in Shorthand. Appendix E gives the complete classification. An invalid answer can reflect syntax, arm-adherence, type conversion or static-analysis restrictions. We do not classify every analysis rejection as an algorithmic inability, nor do we infer a single cause from the first diagnostic.

6 Operational provenance and audit

Sonnet spans nine registered segments. The initial shared queue generates 119 calls at concurrency four. Its next seven segments generate 76, 24, 88, 80, five, 97 and 15 calls at concurrency one. The final segment generates

35 calls at concurrency four. These counts reconcile to 539 cumulative calls and are not additional batches of initial answers.

The user reports account labels A, B and C. Before the B login, A has 297 initial answers; B completes the remaining 48 and 28 repairs before the C login. The recorded sign-in boundaries are user-reported, not independently verified account identities. The study preserves this variation as a threat to within-model operational consistency.

Host sleep on 2 October from 11:30:59 to 11:50:48 PKT affects an active Sonnet transport. Its eventual reply reports a network issue and known usage of 7,156 input and three output tokens. The provider duration is about 1,287 seconds, while local monotonic transport duration is about 235 seconds. An earlier diagnostic ledger incorrectly described this as a timeout with unknown usage; an appended correction supersedes it. We preserve both records and do not treat that wall time as uninterrupted model latency.

Quota replies include a historical reset of 153h5m36s and a later reset of 1h42m17s at 13:05:59 UTC. The frozen supervisor selects a maximum historical deadline and incorrectly schedules 8 October. User-authorized saved-state corrections supply earlier retry times while retaining the old messages and frozen controller. The last corrected deadline is 14:49:10 UTC, and controller polling launches at 14:50:01.958 UTC. A separate interrupted-stream error also remains in the failure ledger. None of these messages independently establishes a weekly cap or account identity.

At the user’s request, the last one-worker Sonnet segment stops submitting new calls and lets its active answer finish. It saves that answer at 15:42:36.971 UTC and exits normally. A separate frozen continuation starts at 15:44:23.805 UTC with concurrency four. After a usable real-task access gate, the audit verifies four concurrent calls. The segment completes at 16:11:31.944 UTC. The change affects throughput and exposure to rate limits; no prior cell repeats and no frozen source changes.

Sol first makes two unsupported-model attempts using CLI 0.146.0. The user updates to CLI 0.160.0 and authorizes a retry; the first real pending task succeeds and releases the access hold. A later capacity error is retained and recovered without changing the model. All three failures have unknown usage. The last automatic continuation inherits stale 0.146.0 metadata although the actual client is 0.160.0. We preserve the frozen manifest and record an external version erratum. All 409 traces remain in the audit.

The final audit verifies all registered frozen artifacts, exact initial prompt parity, complete cell keys, round order, no repair after success, no scoring failures, correct-output cardinality and raw-call reconciliation. Latest results are archived byte for byte. The complete failure ledger retains provider errors and usage without hiding unknown values. The original paper, Git status and binary diff against HEAD match saved baselines before this separate manuscript is added.

7 Limits and future work

The corpus is small, selected for checked representability, and dominated by string recurrences. HumanEval+ and MBPP+ contracts are adapted. Four simple MBPP problems have strong structural shortcuts, and synthetic authorship does not prove novelty to the models. Finite tests cannot establish all-input task correctness.

Rust is more expressive than Intent, and the language arms differ in guides, schema inference, parser conventions and static obligations. Core versus Shorthand controls more of these factors, but the experiment cannot isolate syntax from model familiarity or instruction interpretation. Model-specific training data and hidden client context are unknown. Repeated trials may correlate. Exploratory uncertainty estimates do not justify population-wide conclusions.

Provider token conventions, caches and reasoning categories differ. No prices, energy measurements, compute counters or stable comparable latency measurements are available. Account changes, host sleep, quota interruptions, a CLI update and the final concurrency change further limit throughput and cross-model comparisons. A zero reported thinking category cannot prove zero reasoning, and unknown failed-call usage cannot be replaced by zero.

The next controlled study should hold model and effort fixed, compare paired prompt guides, include more independently authored tasks and evaluate notation-specific training. It should measure correct equivalent program lengths separately from all emitted programs and record complete provider usage. A later language with direct execution could test whether Rust transpilation contributes overhead. These are future experiments, not demonstrated benefits of the present encodings.

8 Conclusion

Equivalent checked references confirm that the Shorthand schema reduces program text. Existing hosted models do not turn that representational saving into a consistent advantage in successful generation. Gemini, Sonnet and Luna solve fewer repetitions with Shorthand than Core and spend more reported output per solved repetition. Sol’s initial accuracy is close, while incomplete failed-call usage leaves its full cost comparison unknown. Evaluating compact programming interfaces therefore requires separate measurements of program length, functional success, bounded repairs and complete call accounting.

A Complete task corpus

Task	Function	Cases
HumanEval/18	how_many_times	1004
HumanEval/23	strlen	967
HumanEval/48	is_palindrome	996
HumanEval/54	same_chars	1006
HumanEval/80	is_happy	949
HumanEval/82	prime_length	978
Mbpp/79	word_len	105
Mbpp/238	number_of_substrings	105
Mbpp/242	count_charac	106
Mbpp/247	lps	107
Mbpp/741	all_Characters_Same	104
Fresh/01	pair_match_count	574
Fresh/02	pair_order_count	574
Fresh/03	alternating_triples	370
Fresh/04	equal_nonoverlapping_pairs	370
Fresh/05	local_peaks	370
Fresh/06	reverse_aligned_matches	574
Fresh/07	matching_at_even_positions	574
Fresh/08	unequal_mirror_pairs	370
Fresh/09	longest_run	370
Fresh/10	longest_common_substring	574
Fresh/11	weighted_alignment_cost	574
Fresh/12	weighted_lcs	574

Table 7: The complete selected corpus. HumanEval+ contributes 5,900 cases, MBPP+ 527 and synthetic tasks 5,868. Case counts are per task rather than multiplied by repeated trials or repair rounds.

Every selected task has a checked Core and Shorthand reference. Their complete case outputs and canonical digests agree. The synthetic labels identify study-authored tasks; no official benchmark label applies to them.

B Input, cache and unknown usage

Model	Arm	Input	Cache read	Missing I	Missing cache write
Gemini low	Rust	919,618	24,396	0	72
	Rust + library	1,019,670	12,201	0	75
	Core	1,347,016	12,204	0	99
	Shorthand	1,722,928	24,408	0	119
	Shape	1,522,233	24,414	0	112
Sonnet	Rust	297,163 [†]	700,419 [†]	1	72
	Rust + library	342,794	720,721	0	73
	Core	508,899	903,439	0	113
	Shorthand	687,692	1,211,545	0	155
	Shape	604,531	1,045,553	0	126
Luna high	Rust	1,439,543	1,185,280	0	0
	Rust + library	1,227,400	934,400	0	0
	Core	1,709,377	1,338,112	0	0
	Shorthand	2,338,437	1,795,840	0	0
	Shape	2,496,405	1,920,000	0	0
Sol medium	Rust	1,512,079	1,199,360	0	0
	Rust + library	1,427,555	1,112,448	0	0
	Core	1,812,321 [†]	1,351,936 [†]	3	3
	Shorthand	1,804,726	1,426,048	0	0
	Shape	2,014,402	1,558,272	0	0

Table 8: Complete input and cache accounting by arm. A dagger marks an available lower bound. Cache read counts remain separate from reported input counts because provider conventions differ. Codex reports zero cache writes on known calls. The Antigravity replies omit cache-write usage, so every such call has unknown cache-write usage.

Across arms, Gemini reports 6,531,465 input tokens, 1,494,421 output tokens and 1,154,626 thinking tokens. Luna reports 9,211,162 input, 903,601 output and 847,812 thinking tokens. Sonnet’s available totals are 2,441,079 input and 1,326,894 output, with one unknown call. Sol’s available totals are 8,571,083 input, 120,635 output and 80,698 thinking, with three unknown calls. These totals retain infrastructure attempts and are not comparable billing amounts.

C Task-by-scheme thinking and output usage

Figures 4–7 show exact counts for all 460 model/task/scheme combinations. R is standalone Rust, L is Rust with a library, C is Core, S is Shorthand and G is Shape. HE denotes adapted HumanEval+, M denotes adapted MBPP+, and F denotes a study-authored Fresh task. Table 7 supplies full task names. Each cell includes the three repetitions and all associated repairs and infrastructure attempts. Numbers are totals in tokens, not thousands.

Blue encodes available output and orange encodes available thinking. Within each metric, the same logarithmic color scale applies across models. Color intensity is proportional to $\log_{10}(1 + v)$, with zero white and the largest available value at 85% of the base color. Numeric labels retain exact counts. A dagger means an available lower bound because one or more calls lack that usage field. Gray 0* cells are Sonnet’s reported zero thinking category and must not be read as measured absence of reasoning. Output already contains thinking, so the two maps cannot be summed.

Total output O					
Task	R	L	C	S	G
HE/18	1141	1012	25830	42916	51922
HE/23	863	214	10111	3545	2139
HE/48	564	820	10145	1860	1381
HE/54	906	1516	102537	166172	101589
HE/80	964	858	3286	10555	4713
HE/82	1219	1106	17794	34595	23366
M/79	409	558	5129	11588	2676
M/238	1467	1693	4253	4862	3300
M/242	1043	788	13135	3206	820
M/247	4193	6222	10094	4086	19196
M/741	1173	1109	12003	6503	2396
F/01	1509	752	2721	317	7585
F/02	3383	1433	13420	4759	13646
F/03	960	1371	3525	505	11860
F/04	1325	592	13713	15416	9357
F/05	3154	1957	8835	3595	6090
F/06	2140	784	14856	41539	215
F/07	1513	1038	17658	34291	14129
F/08	1339	871	12744	5926	9887
F/09	1904	483	12715	50248	94032
F/10	1477	1332	47902	61239	50534
F/11	2782	1098	10784	28181	59132
F/12	3635	2541	9359	6641	10151

Reported thinking H					
Task	R	L	C	S	G
HE/18	832	682	24919	12916	45357
HE/23	746	97	9961	3422	2083
HE/48	306	562	9877	1647	1301
HE/54	501	1117	72435	95828	55383
HE/80	579	143	2791	9642	4371
HE/82	836	661	16927	34167	17668
M/79	255	420	4917	11426	2618
M/238	1284	1510	3996	4709	3238
M/242	917	662	12906	3057	769
M/247	3458	5897	9670	3785	12716
M/741	893	729	11718	6267	2331
F/01	1008	351	2291	0	7516
F/02	2673	916	12991	4450	13564
F/03	615	1069	3095	0	11740
F/04	1086	327	8774	15169	7034
F/05	2892	1695	8517	3245	5970
F/06	1764	349	14211	40903	0
F/07	1176	672	12283	33353	13994
F/08	1035	568	11929	5478	6111
F/09	1419	0	11529	49050	76808
F/10	721	731	35944	43651	24745
F/11	2207	569	9962	27181	41271
F/12	2986	1881	8905	6304	10043

R: Rust L: Rust + library C: Core S: Shorthand G: Shape

Figure 4: Gemini low. Exact output and reported thinking by task and scheme, summed over the complete bounded protocol. Both usage fields are known for every retained call.

Total output O						Reported thinking H					
Task	R	L	C	S	G	Task	R	L	C	S	G
HE/18	465	613	69721	31712	35560	HE/18	0*	0*	0*	0*	0*
HE/23	352	276	1259	1072	1058	HE/23	0*	0*	0*	0*	0*
HE/48	465	1858	11443	13755	1864	HE/48	0*	0*	0*	0*	0*
HE/54	681	1240	0	0	0	HE/54	0*	0*	0*	0*	0*
HE/80	537	650	17962	14170	18333	HE/80	0*	0*	0*	0*	0*
HE/82	453	451	41006	60637	33223	HE/82	0*	0*	0*	0*	0*
M/79	617	669	1746	2139	1171	M/79	0*	0*	0*	0*	0*
M/238	613	540	1789	1809	3681	M/238	0*	0*	0*	0*	0*
M/242	338	318	1304	2127	888	M/242	0*	0*	0*	0*	0*
M/247	1425	927	23504	47634	52037	M/247	0*	0*	0*	0*	0*
M/741	672	1311	6355	3309	1810	M/741	0*	0*	0*	0*	0*
F/01	657	1129	8139	3830	2957	F/01	0*	0*	0*	0*	0*
F/02	772	1969	8579	3570	4485	F/02	0*	0*	0*	0*	0*
F/03	577 [†]	704	5866	10370	9514	F/03	0* [†]	0*	0*	0*	0*
F/04	457	570	51711	11376	17057	F/04	0*	0*	0*	0*	0*
F/05	753	821	8598	14963	40798	F/05	0*	0*	0*	0*	0*
F/06	614	601	8104	12231	2585	F/06	0*	0*	0*	0*	0*
F/07	615	890	36764	26848	12809	F/07	0*	0*	0*	0*	0*
F/08	533	667	17229	29006	10035	F/08	0*	0*	0*	0*	0*
F/09	639	6740	46593	27149	49840	F/09	0*	0*	0*	0*	0*
F/10	952	2277	31593	67702	27114	F/10	0*	0*	0*	0*	0*
F/11	658	1994	35157	51665	45658	F/11	0*	0*	0*	0*	0*
F/12	1813	4802	9169	6454	19623	F/12	0*	0*	0*	0*	0*

R: Rust L: Rust + library C: Core S: Shorthand G: Shape

Figure 5: Sonnet. Exact output and reported thinking by task and scheme, summed over the complete bounded protocol. The Fresh/03 Rust cell has one unknown call. All known thinking fields are zero; gray 0* records that reporting convention.

Total output O						Reported thinking H					
Task	R	L	C	S	G	Task	R	L	C	S	G
HE/18	519	656	12279	30653	23384	HE/18	314	421	11731	29936	22746
HE/23	709	316	824	644	488	HE/23	543	211	638	531	431
HE/48	940	567	3818	18812	1227	HE/48	663	366	3364	18197	1141
HE/54	551	871	51071	70378	44353	HE/54	240	550	46998	67509	42821
HE/80	1180	660	4145	4909	7049	HE/80	792	339	3590	4317	6408
HE/82	1070	736	6987	25297	19445	HE/82	755	420	6425	24639	19040
M/79	1123	616	1703	1516	1432	M/79	890	487	1508	1384	1341
M/238	1163	1258	1665	1580	1297	M/238	800	987	1467	1448	1234
M/242	865	958	842	1855	783	M/242	565	784	645	1659	720
M/247	2426	1754	13565	13717	15070	M/247	1364	1203	12531	13008	14779
M/741	1952	1048	1905	2370	908	M/741	1421	674	1647	2124	830
F/01	1008	796	2262	1541	10858	F/01	672	559	1890	1290	10507
F/02	1677	1473	4230	1612	4405	F/02	844	1031	3846	1359	4207
F/03	795	740	1696	2481	3900	F/03	449	509	1410	2232	3587
F/04	1322	642	10874	13996	2281	F/04	985	418	10275	13562	2143
F/05	1159	726	2524	11434	5316	F/05	835	438	2238	11047	4992
F/06	855	522	2415	34427	6313	F/06	504	255	2006	33594	5945
F/07	938	563	6873	24453	9503	F/07	606	317	6292	23248	9103
F/08	755	637	4919	17837	6632	F/08	421	403	4386	17211	6271
F/09	954	720	31739	31110	39021	F/09	529	343	30629	30006	38441
F/10	1483	1441	34165	73592	30275	F/10	909	980	32583	72458	29634
F/11	793	693	6150	8504	5634	F/11	344	354	4954	7460	5281
F/12	2510	1203	3598	3085	11632	F/12	1443	644	3180	2807	11370

R: Rust L: Rust + library C: Core S: Shorthand G: Shape

Figure 6: Luna high. Exact output and reported thinking by task and scheme, summed over the complete bounded protocol. Both usage fields are known for every retained call.

Total output O						Reported thinking H					
Task	R	L	C	S	G	Task	R	L	C	S	G
HE/18	352	322	1660 [†]	3909	3150	HE/18	50	109	1173 [†]	3501	2700
HE/23	177	104	483	253	196	HE/23	0	0	265	142	157
HE/48	261	123	469	352	146	HE/48	50	0	244	199	67
HE/54	482	303	9306	6847	12577	HE/54	95	0	6337	4819	11104
HE/80	397	291	730	882	1064	HE/80	20	0	388	600	853
HE/82	582	343	1006	1333	1139	HE/82	136	0	679	1155	985
M/79	231	117	630	80	203	M/79	31	0	412	0	162
M/238	511	548	367	280	162	M/238	247	366	172	151	99
M/242	230	105	636	150	279	M/242	37	0	401	58	234
M/247	735	583	1272	1239	389	M/247	170	409	798	984	320
M/741	319	398	562	592	196	M/741	71	236	324	435	118
F/01	585	371	672	566	403	F/01	210	109	327	323	343
F/02	940	719	844	529	411	F/02	386	367	499	286	342
F/03	390	291	445 [†]	1131	385	F/03	40	88	184 [†]	891	280
F/04	527	180	650	972	378	F/04	143	0	396	767	258
F/05	496	286	515	1241	515	F/05	208	86	254	1001	385
F/06	414	180	584	1722	306	F/06	141	0	287	1506	201
F/07	497	213	774	1781	426	F/07	104	0	464	1521	312
F/08	496	232	832	1011	1185	F/08	164	0	517	821	941
F/09	483	312	965	4883	3293	F/09	73	0	552	4240	2855
F/10	2558	726	3223	7277	2046	F/10	547	198	1778	6033	1360
F/11	573	459	1389	2432	1357	F/11	146	121	956	2114	1080
F/12	1165	785	607	515	439	F/12	372	235	229	242	352

R: Rust L: Rust + library C: Core S: Shorthand G: Shape

Figure 7: Sol medium. Exact output and reported thinking by task and scheme, summed over the complete bounded protocol. Core has two unknown calls on Fresh/03 and one on HumanEval/18. Dagged cells are available lower bounds; complete usage is unknown.

D Code-only tokens by task and scheme

Figures 8–11 measure extracted program text with the common auxiliary tokenizer `o200k_base`. Each left-hand cell sums the three initial programs for one task and scheme. Dividing that cell by three gives the mean initial code length for that task. Each right-hand cell adds every usable repair program for the same repetitions. It includes the initial programs, so the two panels must not be added. Their difference is the extracted code in the repairs.

The maps exclude surrounding response prose and reasoning. They retain incorrect and invalid programs. Failed transports are outside this extracted-program measurement; their reported usage and missing fields remain in the separate provider accounting. All 1,380 initial programs and 550 usable repair programs have saved code counts, independently rechecked against the extracted text. These counts do not fill the four missing provider-usage records.

R denotes standalone Rust, L Rust with a library, C Core, S Shorthand and G Shape. HE, M and F retain the task labels from Appendix A. Every cell prints an exact total in auxiliary tokens. Both panels and all four models use the same green logarithmic scale, proportional to $\log_{10}(1 + v)$, with zero white and the maximum at 85% green. These scales differ from the output and thinking maps. The initial panel compares the same number of programs in every cell; the cumulative panel also reflects differing repair counts. Neither panel measures code per correct solution or provider billing.

Initial code, three answers					
Task	R	L	C	S	G
HE/18	236	255	568	239	78
HE/23	87	87	114	91	41
HE/48	210	210	211	165	64
HE/54	321	315	1017	813	151
HE/80	316	296	420	355	195
HE/82	317	373	418	199	134
M/79	121	105	104	47	44
M/238	144	144	152	114	40
M/242	93	93	140	114	33
M/247	616	262	337	226	99
M/741	230	228	232	125	60
F/01	384	311	343	248	48
F/02	469	397	342	234	66
F/03	282	248	258	204	96
F/04	193	214	280	199	111
F/05	218	218	258	203	107
F/06	170	171	319	176	91
F/07	258	288	360	280	111
F/08	245	243	265	177	91
F/09	369	367	536	173	276
F/10	611	481	676	477	278
F/11	443	408	418	287	124
F/12	530	535	367	261	87

Code through all repairs					
Task	R	L	C	S	G
HE/18	236	255	747	579	243
HE/23	87	87	114	91	41
HE/48	210	210	211	165	64
HE/54	321	315	3416	1830	637
HE/80	316	296	420	480	303
HE/82	317	373	744	350	277
M/79	121	105	163	113	44
M/238	144	144	203	114	40
M/242	93	93	180	114	33
M/247	616	262	337	226	99
M/741	230	228	232	183	60
F/01	384	311	343	248	48
F/02	469	397	342	234	66
F/03	282	248	258	402	96
F/04	193	214	579	199	178
F/05	218	218	258	280	107
F/06	294	339	521	402	183
F/07	258	288	839	726	111
F/08	245	243	655	347	205
F/09	369	367	1019	636	449
F/10	611	481	1065	1287	673
F/11	443	408	668	805	301
F/12	530	535	367	261	87

R: Rust L: Rust + library C: Core S: Shorthand G: Shape

Figure 8: Gemini low. Code-only totals by task and scheme. The left panel includes three initial programs per cell; the right includes those programs and all usable repairs. Counts exclude reasoning and response prose and include invalid or incorrect programs.

Initial code, three answers

Task	R	L	C	S	G
HE/18	291	291	480	111	94
HE/23	110	84	159	48	23
HE/48	181	205	271	158	63
HE/54	382	391	0	0	0
HE/80	327	311	391	343	256
HE/82	264	249	399	270	116
M/79	123	157	184	88	31
M/238	228	141	182	80	28
M/242	90	90	177	30	31
M/247	715	145	389	245	86
M/741	297	270	231	150	60
F/01	317	347	351	231	56
F/02	362	718	347	233	48
F/03	303	315	299	210	96
F/04	255	255	313	118	103
F/05	315	315	321	248	140
F/06	296	265	292	153	91
F/07	356	358	378	231	100
F/08	240	240	304	192	96
F/09	411	381	451	219	245
F/10	650	557	173	265	122
F/11	396	617	530	268	111
F/12	655	863	387	261	80

Code through all repairs

Task	R	L	C	S	G
HE/18	291	291	1005	320	199
HE/23	110	84	245	59	23
HE/48	181	205	678	269	63
HE/54	382	391	0	0	0
HE/80	327	311	391	468	256
HE/82	264	249	682	637	188
M/79	123	157	184	126	31
M/238	228	141	182	150	46
M/242	90	90	177	95	31
M/247	715	145	639	502	311
M/741	297	353	231	150	60
F/01	317	347	351	231	90
F/02	362	718	347	233	93
F/03	303	315	299	360	139
F/04	255	255	531	305	138
F/05	315	315	321	592	228
F/06	296	265	292	564	91
F/07	356	358	564	721	137
F/08	240	240	505	556	218
F/09	411	381	1028	808	337
F/10	650	557	631	506	195
F/11	396	775	1353	919	367
F/12	655	1179	387	261	185

R: Rust L: Rust + library C: Core S: Shorthand G: Shape

Figure 9: Sonnet. Code-only totals by task and scheme. The left panel includes three initial programs per cell; the right includes those programs and all usable repairs. Counts exclude reasoning and response prose and include invalid or incorrect programs.

Initial code, three answers					
Task	R	L	C	S	G
HE/18	187	217	530	328	186
HE/23	113	87	168	95	39
HE/48	259	183	253	196	68
HE/54	293	303	1812	976	712
HE/80	370	303	372	304	204
HE/82	297	298	324	263	164
M/79	106	111	177	114	46
M/238	174	170	180	114	45
M/242	87	119	179	86	45
M/247	488	533	382	257	83
M/741	215	212	240	164	60
F/01	318	219	354	233	85
F/02	494	351	366	235	73
F/03	328	213	268	231	160
F/04	319	206	294	194	120
F/05	306	270	268	217	134
F/06	333	249	391	258	141
F/07	314	228	454	343	97
F/08	316	216	306	200	116
F/09	407	359	497	275	190
F/10	556	443	443	283	103
F/11	431	321	340	242	93
F/12	616	541	400	260	76

Code through all repairs					
Task	R	L	C	S	G
HE/18	187	217	530	681	584
HE/23	142	87	168	95	39
HE/48	259	183	424	567	68
HE/54	293	303	4031	2815	1496
HE/80	370	303	531	562	593
HE/82	297	298	532	616	363
M/79	203	111	177	114	67
M/238	333	247	180	114	45
M/242	264	150	179	166	45
M/247	1026	533	698	667	249
M/741	489	344	240	222	60
F/01	318	219	354	233	297
F/02	803	418	366	235	162
F/03	328	213	268	231	283
F/04	319	206	569	398	120
F/05	306	270	268	357	288
F/06	333	249	391	779	332
F/07	314	228	557	1151	352
F/08	316	216	503	578	313
F/09	407	359	1074	1050	526
F/10	556	443	1540	1080	587
F/11	431	321	1142	761	299
F/12	1037	541	400	260	214

R: Rust L: Rust + library C: Core S: Shorthand G: Shape

Figure 10: Luna high. Code-only totals by task and scheme. The left panel includes three initial programs per cell; the right includes those programs and all usable repairs. Counts exclude reasoning and response prose and include invalid or incorrect programs.

Initial code, three answers					
Task	R	L	C	S	G
HE/18	286	195	469	304	131
HE/23	165	92	48	24	21
HE/48	195	111	207	135	63
HE/54	369	291	939	1011	453
HE/80	363	279	324	264	135
HE/82	428	331	309	160	136
M/79	186	105	89	68	23
M/238	246	164	177	111	45
M/242	177	93	54	52	27
M/247	547	156	342	237	51
M/741	232	144	220	139	60
F/01	357	244	327	225	42
F/02	396	334	327	225	51
F/03	242	189	243	222	87
F/04	186	168	236	187	102
F/05	270	186	243	222	112
F/06	255	168	279	198	87
F/07	277	201	292	242	96
F/08	314	220	297	172	102
F/09	394	300	395	257	200
F/10	1993	510	807	501	313
F/11	411	320	415	300	76
F/12	775	532	360	255	69

Code through all repairs					
Task	R	L	C	S	G
HE/18	286	195	469	384	402
HE/23	165	92	188	81	21
HE/48	195	111	207	135	63
HE/54	369	291	2921	1992	1425
HE/80	363	279	324	264	187
HE/82	428	331	309	160	136
M/79	186	105	188	68	23
M/238	246	164	177	111	45
M/242	177	93	203	74	27
M/247	547	156	450	237	51
M/741	232	144	220	139	60
F/01	357	244	327	225	42
F/02	532	334	327	225	51
F/03	332	189	243	222	87
F/04	358	168	236	187	102
F/05	270	186	243	222	112
F/06	255	168	279	198	87
F/07	373	201	292	242	96
F/08	314	220	297	172	208
F/09	394	300	395	601	402
F/10	1993	510	1415	1202	650
F/11	411	320	415	300	235
F/12	775	532	360	255	69

R: Rust L: Rust + library C: Core S: Shorthand G: Shape

Figure 11: Sol medium. Code-only totals by task and scheme. The left panel includes three initial programs per cell; the right includes those programs and all usable repairs. Counts exclude reasoning and response prose and include invalid or incorrect programs.

E Initial answer classification

Model	Arm	Correct	Invalid	Valid but wrong
Gemini low	Rust	67	2	0
	Rust + library	66	3	0
	Core	47	19	3
	Shorthand	39	26	4
	Shape	43	24	2
Sonnet	Rust	69	0	0
	Rust + library	66	0	3
	Core	42	25	2
	Shorthand	25	41	3
	Shape	35	30	4
Luna high	Rust	51	12	6
	Rust + library	64	2	3
	Core	47	12	10
	Shorthand	31	25	13
	Shape	28	36	5
Sol medium	Rust	64	5	0
	Rust + library	69	0	0
	Core	55	14	0
	Shorthand	56	13	0
	Shape	52	15	2

Table 9: Classification of each of the 69 usable initial answers per model and arm. Invalid includes syntax, typing, arm-format and static-analysis rejection. A valid program must also pass every case to count as correct.

The full saved machine-readable report also contains per-stratum summaries, task-matched statistics, sensitivity checks, transport failures, all segment counts and hashes, and the CLI version erratum. It is separate from the historical pilot data.

F Reproducibility and artifact availability

The evidence archive records the task contracts, adapted tests, reference programs, exact arm guides and prompts, seeds, frozen source and manifests, client versions, raw replies and per-call usage. The final capture is `final-20261002T161503639329Z`. Each latest model result is captured once, byte for byte; cumulative continuations are not summed with their predecessors. The analysis JSON contains source SHA-256 values, accounting checks and the CLI version erratum. The record is needed to reconstruct usage totals, distinguish missing usage from zero and audit the protocol’s interruptions.

The compiler source revision and runtime limits appear in the main protocol. The frozen compiler binary SHA-256 is

8585223d77e0f328191486cffb51cc7b0
fbf5b89a639aad42ced4814bdbac94a.

The task-by-scheme figures derive directly from the four exact captured result files. Their 460 cells reconcile to the arm totals and all 1,962 raw calls, including the four calls with unknown usage. The revision retains machine-readable provider-usage and code-only tables with their figure aggregation checks. The code-only maps reconcile to all 1,930 usable answers, and every saved code count is rechecked with the auxiliary tokenizer. Later manuscript edits do not alter frozen prompts, results or the original pilot paper.

At the time of this manuscript revision, these artifacts are retained in the local study archive. No public artifact repository or persistent dataset identifier is specified. Reproduction therefore requires access to that archive and compatible compiler dependencies; a hosted model rerun need not reproduce the same samples. The manuscript is standalone LaTeX with inline figure data and references. Native compilation checks the revised source, while the saved preservation audit covers the completed experiment.

References

- [1] Z. Sun, X. Du, Z. Yang, L. Li and D. Lo. AI coders are among us: Rethinking programming language grammar towards efficient code generation. ISSTA, 2024. <https://doi.org/10.1145/3650212.3680347>.
- [2] Z. Sun, C. Yang, X. Du, Z. Yang, L. Li and D. Lo. Token Sugar: Making source code sweeter for LLMs through token-efficient shorthand. ASE, 2025. <https://arxiv.org/abs/2512.08266>.
- [3] S. Liu et al. ShortCoder: Knowledge-augmented syntax optimization for token-efficient code generation. 2026. <https://arxiv.org/abs/2601.09703>.
- [4] M. Sullivan and A. Koller. Accelerating constrained decoding with token space compression. arXiv revision 2, 1 October 2026. <https://arxiv.org/abs/2605.29986v2>.
- [5] J. Austin et al. Program synthesis with large language models. arXiv:2108.07732, 2021. <https://arxiv.org/abs/2108.07732>.
- [6] M. Chen et al. Evaluating large language models trained on code. 2021. <https://arxiv.org/abs/2107.03374>.
- [7] J. Liu et al. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. 2023. <https://arxiv.org/abs/2305.01210>.
- [8] F. Cassano et al. MultiPL-E: A scalable and extensible approach to benchmarking neural code generation. 2022. <https://arxiv.org/abs/2208.08227>.
- [9] S. Hao et al. Training large language models to reason in a continuous latent space. 2024. <https://arxiv.org/abs/2412.06769>.
- [10] OpenAI. tiktoken, version 0.13.0. <https://github.com/openai/tiktoken>.
- [11] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. PLDI, 2013. <https://people.csail.mit.edu/jrk/halide-pldi13.pdf>.
- [12] J. Wei et al. Chain-of-thought prompting elicits reasoning in large language models. NeurIPS, 2022. <https://proceedings.neurips.cc/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html>.
- [13] C. Snell, J. Lee, K. Xu and A. Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. arXiv:2408.03314, 2024. <https://arxiv.org/abs/2408.03314>.
- [14] N. Muennighoff et al. sl: Simple test-time scaling. arXiv:2501.19393, 2025. <https://arxiv.org/abs/2501.19393>.
- [15] A. Madaan et al. Self-Refine: Iterative refinement with self-feedback. NeurIPS, 2023. https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.
- [16] B. Chen et al. CodeT: Code generation with generated tests. arXiv:2207.10397, 2022. <https://arxiv.org/abs/2207.10397>.
- [17] H. Lightman et al. Let’s verify step by step. arXiv:2305.20050, 2023. <https://arxiv.org/abs/2305.20050>.
- [18] Y. Chen et al. Reasoning models don’t always say what they think. arXiv:2505.05410, 2025. <https://arxiv.org/abs/2505.05410>.