Large Binary Models

Language Modeling of Compiled Bytecode, Without Language

Jovonni L. Pharr
Georgia Cyber Warfare Range
info@gacwr.org

September 2026

Abstract

The language model is built upon language. Its alphabet is the tokenized surface form of human prose, and its competence is competence in that surface form. Yet the true substrate of a program is not English; it is compiled binary, a finite alphabet of machine operations. This work asks whether a model can learn to predict and to generate that binary directly, without the intermediary of natural language, and finds that it can. We train a byte-level model directly upon the raw bytecode of compiled Python—the operation bytes themselves, carrying no source text and drawn from an alphabet of two hundred and fifty-eight symbols—and find that models of ten to fifty-one million parameters drive the held-out cross-entropy from the uniform baseline of 8.01 bits per token to 0.561. Fifty-five percent of that stream is a deterministic inline-cache slot, so the primary metric is bits per *informative* token, and the model is placed against a zero-learning model of the instruction format and against `xz`, the strongest classical compressor on this substrate. The generations are scored not by the statistic that their bytes are legal opcodes, but by a structural and execution ladder that runs them in the real virtual machine, and by their verbatim overlap with the training stream. Every number is re-derived from a machine-checked receipt on disk by an accompanying verification script, which fails if a value in the text and the measurement behind it ever disagree. Four results ground the work in execution. First, an execution-grounded evaluation on *real* compiled bytecode rather than on a synthetic grammar: given three quarters of a held-out standard-library function, the model writes a completion which, spliced back into the true code object and executed, reproduces every one of that function's own outputs on 7.55 percent $[5.31, 10.64]$ of the decisive subpopulation — the functions where a degenerate completion cannot succeed and where every control scores exactly zero — and on 11.76 percent of all 425 functions verified byte-absent from the training stream. Second, the paper's own open question—whether the tractability survives an instruction set without CPython's inline caches—is answered: the next-opcode conditional entropy is 3.114 bits on CPython and 3.096 bits on x86-64 machine code, so the regularity is a property of compiled code rather than of one encoding, while the cost of encoding an *instruction* differs across those instruction sets by a factor of two. Third, the bits-per-informative-token metric is measured rather than rescaled: the inline cache is 55.56 percent of the held-out tokens and 0.25 percent of the model's held-out bits. Fourth, greedy decoding on real code survives a median of 2 instructions before its first divergence, which bounds what this model generates unrolled and names the problem this line of work now faces. We argue that a finite and grounded operation alphabet is the natural primitive vocabulary for neural computation, connect the line to the general purpose neural computer, and state plainly the milestone that remains: the generation of programs that not only disassemble but *run*.

Contents

Part I
Preliminaries

1 The State of Today

The contemporary model of code is, almost without exception, a model of *source text*. It is trained on the characters a human types—identifiers, keywords, whitespace, comments—and it is evaluated on its ability to continue that text plausibly. This is a reasonable choice, for source text is what humans read and write, and it is abundant. But it is worth observing that source text is not what a machine executes. Between the text and the execution stands a compiler, and the object the compiler produces—a sequence of operations drawn from a fixed instruction set—is the program in the only sense that matters to the machine. To model the source is to model the human-facing shadow of computation; to model the compiled operations is to model computation itself.

2 Related Work

This work sits at the confluence of several lines: the neural modeling of code as language, the modeling of *compiled* and low-level artifacts, the learning of program execution, program synthesis under execution feedback, and the long-standing identification of sequence prediction with compression. We position our two distinctive contributions against each in turn: (i) modeling a program's *compiled bytecode directly as the token stream*, with no source text, and (ii) grounding evaluation of the model's own generations in an *execution-based correctness oracle* rather than in perplexity or surface match.

2.1 Neural models of code as language

The dominant paradigm treats a program as *source text* and applies the machinery of language modeling to it. Encoder models such as CodeBERT [2] and encoder–decoder models such as CodeT5 [3] pre-train bimodal representations of code and natural language. Decoder-only models scaled this to generation: Codex [4] synthesized functional code from docstrings, and open successors such as StarCoder [6] and Code Llama [7] trained on large public source corpora. AlphaCode [5] reached competitive-programming performance by generating and filtering large candidate populations, and a related strand learns to *translate* between languages via back-translation on source text [8]. In every case the object of study is the human-facing surface form, drawn from a subword vocabulary of tens of thousands of units. Our substrate is the opposite end of the compilation pipeline: the finite alphabet of operation bytes the compiler emits, with no identifiers and no source text at all.

2.2 Modeling compiled and low-level representations

A separate literature learns representations of *compiled* artifacts, almost always for *analysis* rather than generation. Binary code similarity detection embeds functions from control-flow graphs [9] or from jump-aware and instruction-level transformers over disassembled assembly [11, 10]. Neural decompilation maps low-level code back to a higher-level form [13, 14], and Nova [12] is a *generative* language model over assembly trained with a decompilation objective. Closest in spirit to our raw-byte view, MalConv [15] classifies malware by consuming the raw bytes of an entire executable. Two distinctions define our contribution. First, prior work overwhelmingly operates on *textual assembly* (a disassembled, human-readable rendering) or on whole-file bytes for a classification label; we model the raw `co_code` operation-byte stream, and then the full serialized code object, as the generative token stream itself. Second, these systems are discriminative or translational, whereas we ask a model to *generate* novel compiled artifacts that reconstruct and run in the virtual machine.

2.3 Learned execution and neural program interpreters

A complementary program is to learn what code *does*. "Learning to Execute" [16] trained recurrent networks to predict the output of short programs; Neural Programmer-Interpreters [17] learned compositional execution traces; and neural algorithmic reasoning imitates the step-wise behavior of classical algorithms [18]. Within code pre-training, execution-aware methods inject dynamic signal: TRACED [19] pre-trains on execution traces and CodeExecutor [20] predicts execution traces of Python. These works learn a *map from program to behavior*. Ours runs in the opposite direction—generating the program—and it does not *predict* execution; it *invokes* the real CPython virtual machine on the generated code object and checks the result. Learned execution is the sequel, carried in our open problems as differentiable execution.

2.4 Program synthesis with execution and correctness feedback

The synthesis literature has long induced programs as structured objects checked against a specification rather than continued as text. DeepCoder [21] predicts program attributes to guide search; RobustFill [22] learns string programs under noisy I/O; and execution-guided synthesis conditions generation on intermediate states [23]. In the LLM era, correctness signal is folded back into training and decoding: CodeRL [24] optimizes against unit-test outcomes, and self-debugging [25] iterates on execution feedback. Evaluation by *functional correctness*—executing candidates against tests, as with the pass@k of Codex [4] and the filtering of AlphaCode [5]—is now standard. We adopt correctness as the metric of record; execution-grounded evaluation is not itself novel. What differs is the object checked: a reconstructed `types.CodeType` executed under sandboxed resource limits, checked by a *differential oracle* that decompiles the generated operations to an expression and confirms the virtual machine's output equals that expression's value on probe inputs. The oracle re-implements the same instruction semantics in the same language, so what it delivers (Section 10) is *self-consistency*—the bytecode computes what its own disassembly says—rather than correctness against an external statement of intent. We further separate memorization from generation with a *novel-correct* rate—programs that both run and compute what their disassembly says, and whose operation-and-constant signature is absent from the training corpus—rather than novelty-by-held-out-prompt, and we report it against the novelty a memorization-free sampler achieves.

2.5 Compression as a measure of modeling

That a good sequence model is a good compressor is a long-standing thesis, operationalized by the Hutter Prize [28] and revisited by "Language Modeling Is Compression" [26], which frames foundation models directly as lossless compressors; neural compressors such as NNCP [27] achieve state-of-the-art bits-per-byte on text. Our bits-per-token reporting and our comparison against `gzip`, `bzip2`, `zstd` and `xz`/LZMA, an order-two byte n-gram, and—the baseline that turns out to matter most on this substrate—a *zero-learning model of the instruction format itself*, all on the identical held-out bytes, are a direct instance of this framing. The novelty is not the yardstick but the substrate. Section 5.1 separates how much of the resulting compression is a property of compiled programs from how much is a property of one virtual machine's instruction encoding, since a majority of the stream in CPython's is a deterministic placeholder; that separation is the difference between a statement about computation and a statement about a byte layout.

2.6 Small and data-efficient language models

Finally, our models are small—ten to fifty-one million parameters—placing them among efforts to show that competence need not require scale when the data are clean or structured. TinyStories [29] shows sub-ten-million-parameter models producing coherent text from a constrained corpus; the phi line argues that "textbook-quality" data yields strong small models [30]; and SmolLM2 [31] documents data-centric training of a compact model. Our result is a data-side analogue: the compiled substrate is so regular that a small model attains near-deterministic grammar and runnable generation on it, precisely because the alphabet is finite and the grammar rigid.

3 Current Problems

Two problems motivate the direct modeling of compiled binary. The first is a problem of *grounding*. A model of source text learns the statistics of a human notation; it does not, except incidentally, learn the semantics of execution. The compiled operations, by contrast, are the

semantics—each is a defined transformation of machine state—so a model of the operations is a model of what the program does, not merely of how it is written. The second is a problem of *efficiency and regularity.* Source text is drawn from a vocabulary of tens of thousands of subword units and is riddled with the irregularities of human expression; compiled operations are drawn from a fixed alphabet of some hundreds of symbols and obey the rigid grammar of a virtual machine. The substrate is smaller and far more regular, and one may reasonably expect it to be more predictable.

Part II
Proposal

4 The Corpus

Definition 1 (Bytecode corpus)**.** *Compile every available Python source file of a standard installation—the standard library together with installed packages—to a code object. From each code object, recursively extract the raw operation-byte string of every nested code object it contains. Concatenate these strings, delimiting code objects and source files by two reserved symbols. The resulting alphabet is the two hundred and fifty-six byte values together with a code-object separator and a file separator, for a vocabulary of two hundred and fifty-eight symbols.*

Applied to a standard installation, this procedure yields a corpus of 124.2 million tokens over 414,522 code objects (Table 1). No source text enters the model; the model observes only the compiled operations.

The builder globs source files in operating-system directory order and truncates the list to twenty thousand files *before* compiling any of them. The raw corpus contains exactly 20,000 file-separator symbols, which is the truncation bound to the unit: the figure is a cap rather than a measurement of coverage, and any compile-failure count is a property of the prefix that was attempted rather than of the installation. Files beyond the first twenty thousand were never opened. The size of that discarded tail is not recoverable: the same glob on the present interpreter returns 12,328 files, so the corpus cannot be rebuilt from this environment.

Two further properties of the split bound how the held-out cross-entropy of the next section reads. First, the split is positional—the final five percent of the concatenated stream—and is therefore neither content-aware nor module-aware. Second, the corpus is not deduplicated. Measuring on the corpus itself: 35.56 percent of training code objects are exact byte-duplicates of another training code object, and 36.97 percent of held-out code objects have a `co_code` byte string that also occurs verbatim in the training split, covering 10.88 percent of held-out tokens. Restricting to non-trivial objects of at least forty bytes, 22.87 percent are still exact duplicates, and a further 17.77 percent lie within a normalised edit distance of 0.2 of some training object. A substantial part of the held-out number reported below is therefore retrieval.

A second builder deduplicates code objects globally, holds out whole top-level packages instead of a positional tail, and records rather than truncates its file counts. On that rebuilt corpus, exact held-out contamination falls to zero. Two findings from the rebuild are worth stating because they bound what the fix buys: holding out whole packages *alone* leaves 29.63 percent exact contamination—it is the deduplication that does most of the work—and near-duplication survives either measure, with 23.95 percent of held-out objects still within edit distance 0.2 of a training object. Retraining on the clean corpus is queued; the models reported here are the models trained on the positional split, and every number below is to be read with that split in mind.

Table 1: The compiled-bytecode corpus and the three trained models. Parameter counts are the ones each run's own log prints. The reported bits-per-token estimator is the mean and standard deviation of the last twenty held-out evaluations of each run; the minimum over the roughly one hundred single-batch evaluations of the same run is a selection-over-noise statistic and is shown in the final row for completeness rather than reported as the result. The uniform baseline is $\log_2 258 = 8.01$. Every value is re-derived from its receipt by the verification pass.

Corpus		**Models (width / depth)**			
tokens	124.2M	metric	d320/L8	d448/L12	d512/L16
code objects	414,522	parameters	10.2M	29.4M	51.0M
file separators	20,000	init. bits/token	8.01	8.01	8.01
train dup. objects	35.56%	bits/token (last 20)	0.610	0.574	**0.561**
held-out exact dup.	36.97%	standard dev.	0.042	0.047	0.032
held-out near dup.	17.77%	epochs over train	2.22	2.25	2.86
vocabulary	258	*single-batch* min	0.531	0.480	0.460

5 The Model and Its Behavior

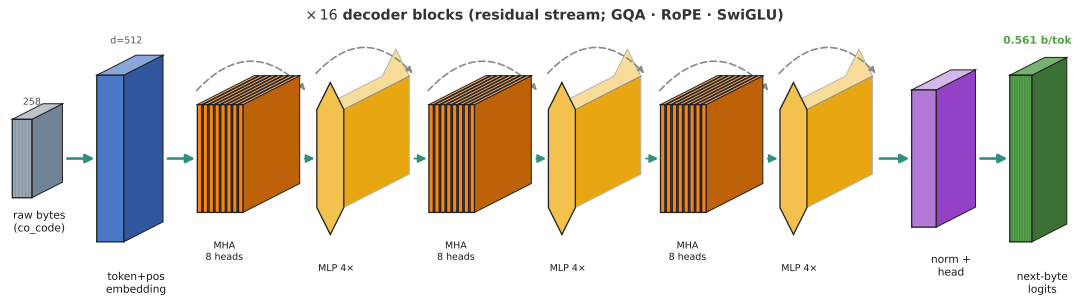

Figure 1: **Large Binary Models: a byte-level decoder-only Transformer over compiled CPython bytecode.** The alphabet is the two hundred and fifty-eight raw operation-byte symbols. No natural language enters the model; it consumes compiled bytecode and predicts the next operation byte, driving held-out cross-entropy from the uniform 8.01 bits/token to 0.561.

The model is a standard decoder-only Transformer over the byte alphabet, of ten to fifty-one million parameters according to width and depth. Training minimizes the mean next-token negative log-likelihood over the operation stream $b_{1:n}$,

$$\mathcal{L}(\theta) \;=\; -\frac{1}{n}\sum_{i=1}^{n} \log_2 p_\theta(b_i \mid b_{<i}) \quad \text{bits/token}, \qquad \mathcal{L}_{\text{uniform}} = \log_2 258 = 8.01, \tag{1}$$

and drives the held-out value from the uniform baseline of 8.01 down to 0.561 bits per token for the largest model—a compression ratio

$$\kappa \;=\; \frac{\mathcal{L}_{\text{uniform}}}{\mathcal{L}(\theta)}, \tag{2}$$

and an indication that compiled binary is highly structured and highly learnable directly, without the mediation of source text (Figure 3).

The estimator behind that number needs stating plainly. The trainer evaluates on a *single random batch* of 32×512 tokens every two hundred steps—one quarter of one percent of the held-out set—so the running *minimum* over roughly one hundred such draws is selection over noise, and it is not comparable to a classical compressor computed over the entire held-out set. Those minima sit between 1.9 and 3.1 standard deviations below the mean of the last twenty evaluations of the same run. Table 1 therefore reports the mean and standard deviation of the last twenty evaluations, 0.561 ± 0.032 for the largest model, and treats 0.460 as the selection statistic it is.

That estimator is checked exhaustively on one model. Scoring the *entire* held-out split of 6.21 million tokens in non-overlapping windows—every token counted exactly once, no sampling at all—the smallest model attains 0.5994 bits per token with a bootstrap 95 percent interval of $[0.5954, 0.6035]$. Its last-twenty mean is 0.610, which lies within a quarter of one standard deviation of that exhaustive value, while its single-batch minimum of 0.531 lies far outside it. The last-twenty mean is therefore a sound stand-in for the exhaustive number and the minimum is not, which is the justification for the Table 1 figures. The two larger models were not scored exhaustively—each is a multi-hour job on a CPU—and that pass is queued.

Scale is measured on the same footing. Thirteen completed runs exist on disk, and across all of them the relation between parameter count and held-out bits per token is *not* monotone: an 18.1M-parameter run reaches 0.452 on the single-batch estimator, below the 51.0M flagship's 0.460, and the best value of any run, 0.432, belongs to a 142.5M model that a 64.4M model fails to match. The entire spread across the thirteen is 1.8 standard deviations of a single evaluation. Figure 4 plots every completed run. The exponent is therefore unmeasured on this record; the study that measures it—a fixed token budget with seeds and confidence intervals—is queued.

What we can say is that the model acquires much of the grammar of the virtual machine: the layout of the operation stream, the sequences of operations that constitute valid computations, and the regularities of their arguments. How much of that is program structure and how much is instruction encoding is the subject of the next section.

5.1 How much of this is the programs, and how much is the instruction format?

The comparison against a uniform prior, while exact, is weak: any model of a structured source beats it. The stronger question has two parts, the alphabet and the choice of classical compressor, and we answer both on the identical held-out bytes.

The first count is the alphabet. Decomposing the held-out stream with the CPython wordcode state machine—each opcode's inline-cache count is a published property of the instruction set—gives opcode slots 22.16 percent of the stream and inline-cache slots 55.23 percent. *Every one* of those cache slots is the byte zero: 100.0000 percent of them, with no exceptions in either split. More than half of the corpus is thus determined by a two-hundred-and-fifty-six-entry lookup table and costs a model that knows the instruction format nothing at all. A hand-written model with *no sequence learning whatsoever*—the format, a unigram over opcode slots, and $P(\text{argument} \mid \text{opcode})$—attains 1.878 bits per token, which already matches the order-two byte n-gram, the obvious learned baseline. Adding an opcode bigram gives 1.496 and a trigram 1.349. This is the baseline the thesis calls for, and it is the one the model must beat.

The second part is the choice of classical compressor. `bzip2` is not the strongest classical compressor on this stream: `xz` at `-9e` attains 1.123 bits per token and, tuned to the stream's two-byte stride, 1.119—both below `bzip2`'s 1.246. The model's single-batch minimum of 0.460 is $2.43\times$ below the strongest classical compressor, and against the last-twenty mean of 0.561 the factor is $2.00\times$.

Because a majority of the stream is free, bits per token is the wrong primary metric, and we replace it with **bits per informative token**: the opcode and argument slots only, 44.3

percent of the stream. On that metric the zero-learning format model costs 4.238, the format-plus-trigram baseline 3.045, `xz -9e` 2.53, and the model 1.038 on the single-batch minimum or 1.265 on the last-twenty mean. That is a factor of roughly two and a half over the strongest classical compressor and of about 2.4 over an opcode trigram, and it is the number to carry forward. It is stated as a rescaling of a cross-entropy measured over all tokens; Section 5.2 measures the per-role decomposition directly with a forward pass and confirms the rescaling to within half a percent.

The reading these two baselines support is precise. A model trained on this substrate learns structure beyond the format—it beats an opcode trigram substantially—and the headline compression ratio is in large part a fact about CPython's inline-cache encoding. On this evidence the tractability of the substrate is a property of the instruction encoding as much as of the programs; the same-size model trained on a cache-stripped stream, which separates the two cleanly, is queued, and Section 7 answers the question at the level of the substrate.

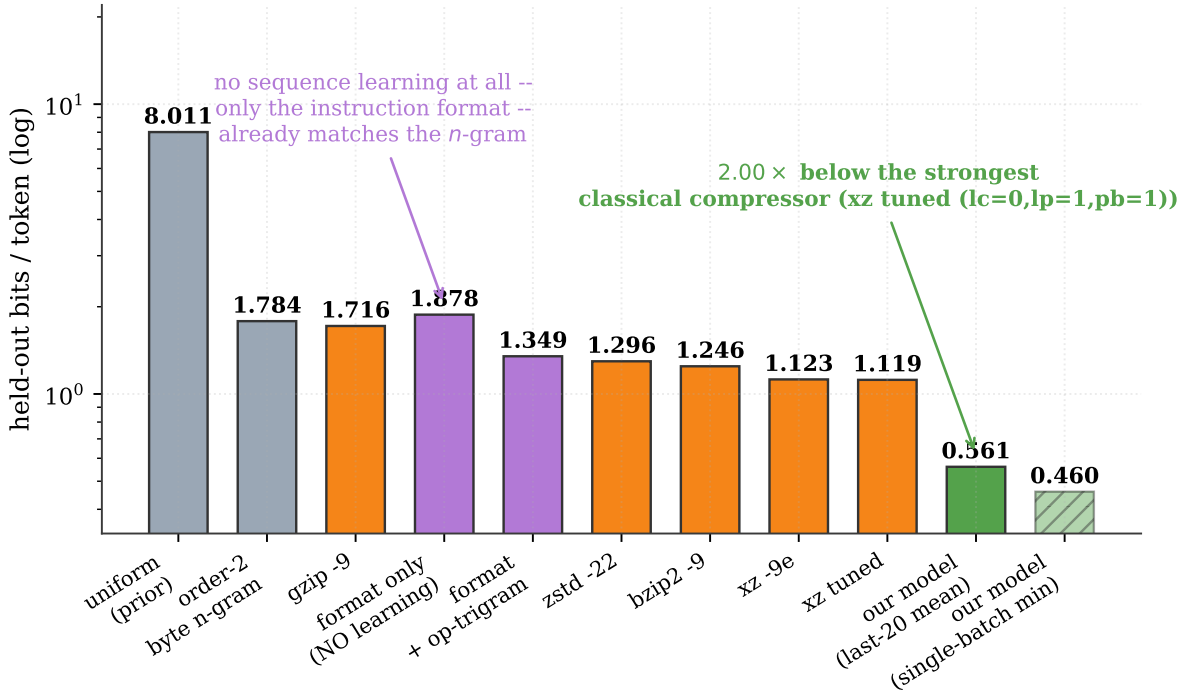

Figure 2: **The full baseline ladder.** Held-out bits per token for every baseline on the identical byte stream, on a logarithmic scale. Two families carry the argument: a zero-learning model of the CPython instruction format (1.878), which knows the wordcode layout and nothing else, with its opcode bigram and trigram extensions (1.496, 1.349); and the `xz`/LZMA compressors (1.123, 1.119), which are the strongest classical compressors on this stream and beat `bzip2` (1.246). The hatched bar is the single-batch minimum, shown for completeness; the reported model value is the last-twenty mean.

The character of what the model *generates* is more telling than the perplexity. Sampled generations, disassembled, are not noise of low perplexity but genuine idioms of real programs. A generation may open with the prologue operation that begins every code object, proceed through the operations that load a global name and then access an attribute upon it—the pattern of a qualified reference such as a call to a library function—and close with the operation that returns a value. The model reproduces even the fine structure of the operation stream, including the approximately correct counts of the inline-cache placeholders that follow certain

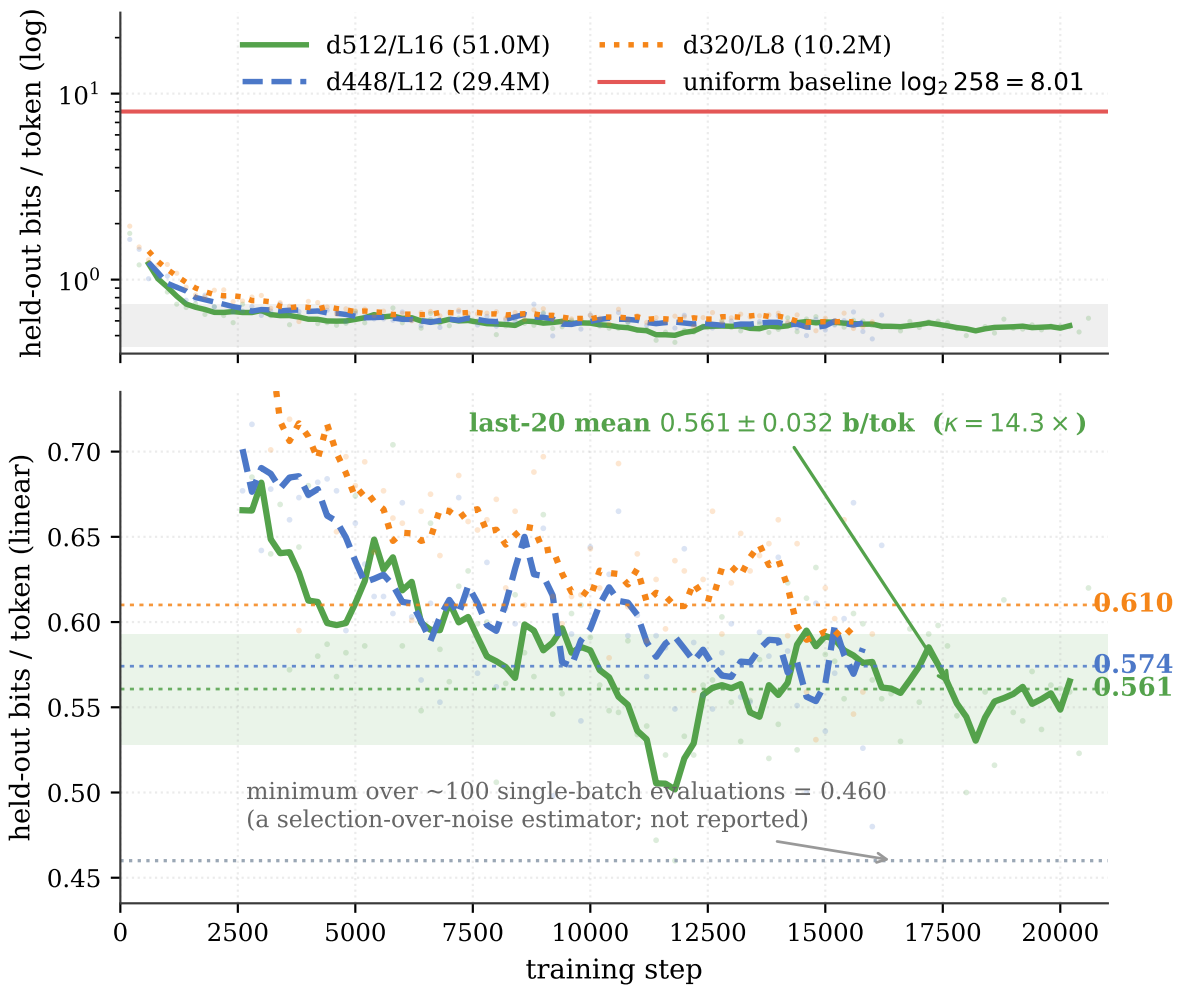

Figure 3: **Learning CPython bytecode directly, with no source text.** Held-out bits per token (Eq. 1) against training step for the three model sizes. *Upper panel:* the whole descent from the uniform baseline of 8.01, on a logarithmic axis. *Lower panel:* the shaded band of the upper panel expanded on a linear axis, where the three sizes separate; each dashed rule is that run's last-twenty mean, and the dotted rule is the single-batch minimum, which lies below every converged curve because it selects over evaluation noise. Faint dots are the raw logged evaluations and the heavy lines are a five-point moving average taken in `"valid"` mode, so every plotted point corresponds to a value that appears in the log and no endpoint is fabricated by zero-padding.

operations in the modern virtual machine.

Whether the model has learned the true distribution over operations, rather than a plausible-looking caricature of it, admits a direct quantitative test. Running that test correctly takes three things that are easy to get wrong on this substrate, and the third decides the answer.

The first is the reference corpus. The comparison must be made against the real held-out five percent of the stream, not against a convenient proxy: importing a list of standard-library modules at figure-build time and walking their function objects reaches modules that are *in the training set*, and yields a different distribution from the recursive module-level compile the corpus builder uses. Measured against the held-out split the statistic is 0.99898; the import-based proxy gives 0.99496.

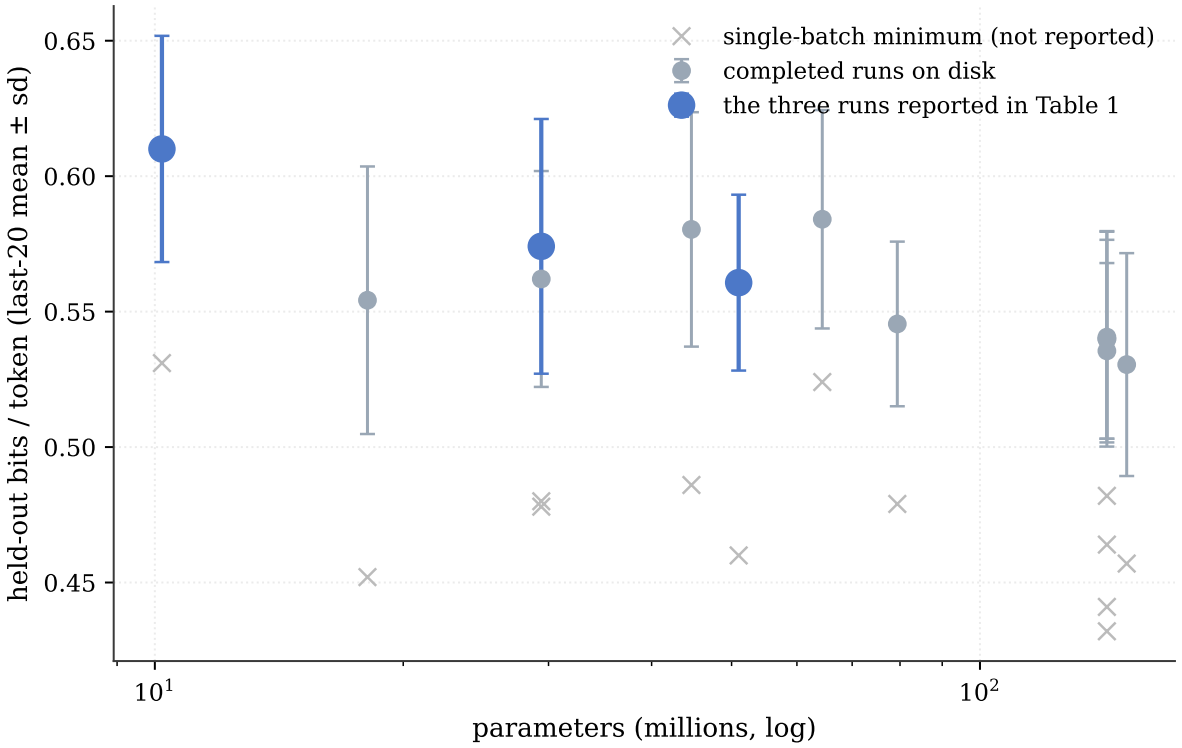

Figure 4: **Every completed run on disk.** Held-out bits per token against parameter count for all thirteen completed runs, with the last-twenty standard deviation as the error bar. The relation is not monotone, and the whole spread is 1.8 standard deviations of a single evaluation, so no fit is made here. Crosses mark the single-batch minimum of each run for reference. A scaling study at a fixed token budget with seeds is queued.

The second is that both sides must be measured identically. We walk both with the CPython wordcode state machine, using each opcode's published inline-cache count. The naive alternative—taking every byte below 256 and then every second one—deletes marker symbols and so shifts the operand/opcode parity of everything after a mid-stream marker. On the generation side that makes no difference here: all twenty-two generations containing a marker carry it at position zero, so no interior marker ever shifts a parity, and the two extractions agree exactly. On the corpus side it makes a large one: the naive extractor inflates the measured cache share from 0.555 to 0.670 and reports every byte value as a distinct opcode.

The third is calibration, and it decides the question. A Pearson correlation over a distribution in which one symbol carries more than half the mass is nearly uninformative until its scale is established, so we measure it on deliberately wrong surrogates. A surrogate that gets *only* the cache placeholder right and assigns zero to every other opcode scores $r = 0.9805$. One that keeps the cache share and randomly permutes every other frequency scores $r = 0.9725$. The entire informative range of the statistic is therefore the interval above 0.98, and a correlation of three nines on this alphabet conveys almost nothing. The decisive null is an i.i.d. resample of the same number of tokens drawn directly from the corpus's own opcode marginals, which by construction has learned nothing but the marginals: it scores $r = 0.999973$ and total-variation distance 0.0119, while the model scores $r = 0.99898$ and total-variation distance 0.0846. **The model's opcode distribution matches the corpus *less* closely than a draw that knows only the marginals does.** A correlation on this alphabet is evidence that one symbol dominates, not evidence of learned structure.

We therefore report total-variation and Jensen–Shannon distance, with bootstrap intervals over generations, on the subalphabet that the instruction format does not determine. Against the real held-out split, at $n = 500$ generations, the total-variation distance is 0.0846 $[0.0765, 0.0953]$ over the full alphabet and 0.1616 $[0.1419, 0.1927]$ with the cache placeholder removed, with a Jensen–Shannon divergence of 0.0460 bits $[0.0410, 0.0594]$. The model overproduces the placeholder (0.5827 of its opcode slots against the corpus's 0.5548) and its generations cover 75 of the 101 opcodes present in held-out code. At the unigram level the model's generations therefore track the corpus but do not match it, and they do not reach the marginals-only null.

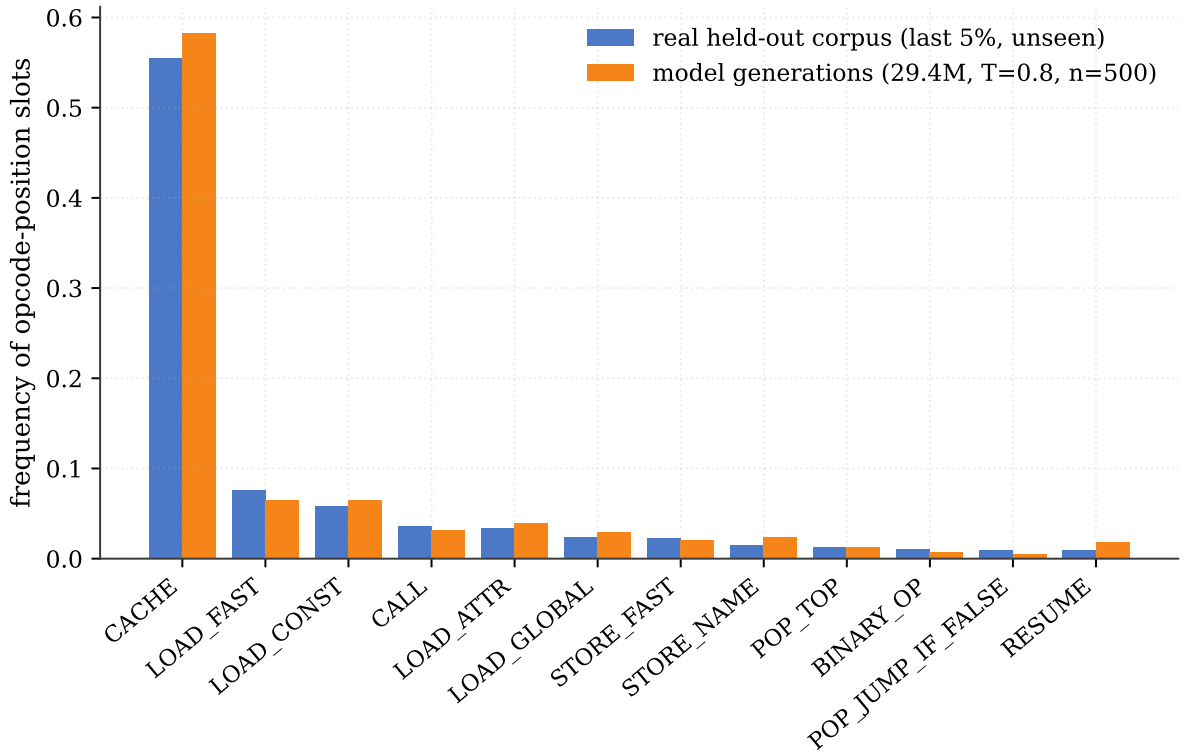

Figure 5: **Generated bytecode against the held-out opcode distribution.** Opcode frequency in the real *held-out* corpus and in the model's own generations, for the twelve most common operations, both sides extracted with the same CPython wordcode state machine. The agreement statistic is the total-variation distance: 0.0846 $[0.0765, 0.0953]$ over the full alphabet and 0.1616 with the cache placeholder removed, against 0.0119 for an i.i.d. resample of the corpus marginals. A Pearson r is near-vacuous on this alphabet: the model scores 0.99898, a surrogate that knows only the cache placeholder scores 0.9805, and the marginal resample scores 0.99997. The cache placeholder is 0.555 of corpus opcode slots against 0.583 of generated ones, and the generations cover 75 of the 101 opcodes present in held-out code.

Formally, writing f_o^{corp} and f_o^{gen} for the held-out and generated frequency of opcode o, the quantity we report is the total-variation distance $\frac{1}{2}\sum_o |f_o^{\text{corp}} - f_o^{\text{gen}}|$, together with its bootstrap interval over generations and the same quantity for the marginal-resampling surrogate that calibrates it. What remains true regardless of how well the model matches it is that the substrate itself is *regular*: the bytecode obeys a near-deterministic grammar in which each operation sharply constrains its successor. Figure 6 exhibits that grammar as the matrix of opcode-to-opcode transition probabilities $P(o_{t+1} \mid o_t)$ measured on the corpus. Its rows are

close to one-hot, and the mean conditional (next-opcode) entropy,

$$H(o_{t+1} \mid o_t) \;=\; -\sum_{o} P(o) \sum_{o'} P(o' \mid o)\, \log_2 P(o' \mid o), \tag{3}$$

is small. Measured on the corpus, it is 1.872 bits on the raw stream the model observes. That number is however dominated by the inline-cache format: on the logical instruction stream, with the cache placeholders removed, the same quantity is 3.122 bits. The low-entropy structure a compact model exploits is therefore substantially the encoding, and only partly the program grammar. The shuffled and argument-randomised controls that separate a learned grammar from the marginals are queued.

The bigram level is in better shape than the unigram level, and it is calibrated the same way. Comparing the transition matrix of the real held-out corpus to that of the model's own generations (Figure 6), with rows normalised over the full alphabet rather than within the displayed fourteen, the Pearson correlation is $r = 0.985$ $[0.9757, 0.9875]$. Stripping the cache placeholder out of the subalphabet entirely, it is $r = 0.940$, against a cache-only surrogate floor of 0.839: unlike the unigram statistic, this one sits meaningfully above its null. The distance that carries the magnitude is the mean per-row total-variation distance, 0.133 $[0.129, 0.160]$, rising to 0.211 on the cache-free subalphabet. The model therefore reproduces the corpus's bigram grammar substantially, and short of identically. We stress that this is a *transition* matrix, not a confusion matrix: it records which operation follows which, so its structure is the columnar and block pattern of the bytecode grammar (every attribute access routing through an inline-cache placeholder, every return preceded by a value load), and no diagonal is expected or meaningful.

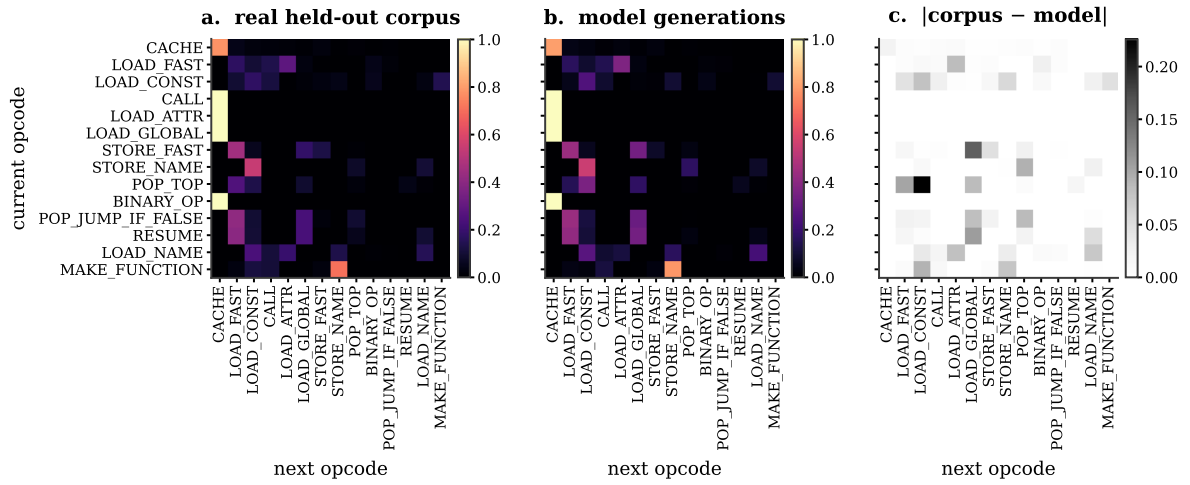

Figure 6: **The opcode transition matrix $P(o_{t+1} \mid o_t)$ over the fourteen most frequent operations.** *(a)* the real held-out corpus grammar; *(b)* the same matrix from the model's own generations; *(c)* their absolute difference on a linear scale, where white is exact agreement. Rows are normalised over the full 256-opcode alphabet, not renormalised inside the displayed fourteen, and the colour scale is linear, with no gamma compression to flatten disagreement. The retained top-fourteen mass is 0.863 of the corpus and 0.902 of the model. Agreement is $r = 0.985$ against a cache-column-only surrogate floor of 0.839, so the distance is the quantity to read: mean per-row total-variation 0.133 $[0.129, 0.160]$, and $r = 0.940$ with the cache column removed. This is a transition matrix, not a confusion matrix—no diagonal is expected; the bright columns are the near-deterministic successors (e.g. every CALL and LOAD_GLOBAL followed by an inline-cache placeholder), the low-entropy regularity of Eq. 3.

5.2 Where the held-out bits actually go

Section 5.1 rescales bits per token to bits per informative token, which assumes something about how the loss distributes across roles. This subsection measures that distribution directly with a forward pass. Every held-out token is labelled OPCODE, ARGUMENT, INLINE-CACHE or MARKER by the same wordcode state machine used above, and the model's bits are accumulated per role over a deterministic, evenly spaced subsample of held-out windows scored end to end—every token in a window counted exactly once.

Table 2: The bit ledger of the held-out split, measured rather than assumed. Scored on 358,400 held-out tokens per model in non-overlapping windows, a deterministic evenly-spaced subsample of the 6.21M-token split; the exhaustive pass over the whole split is queued for the two larger models. The window-level bootstrap intervals on the totals are $[0.5891, 0.6241]$, $[0.5604, 0.5934]$ and $[0.5412, 0.5730]$, and all three contain the Table 1 last-twenty mean of the corresponding model, so this pass independently confirms that estimator on every model.

role	tokens	bits per slot			top-1
	% of stream	10.2M	29.4M	51.0M	51.0M
opcode	22.01	1.322	1.253	1.208	72.77%
argument	22.01	1.417	1.353	1.309	74.33%
inline cache	55.56	0.0037	0.0033	0.0025	100.00%
marker	0.42	0.362	0.338	0.333	93.59%
all tokens	100	0.6063	0.5768	0.5569	—
informative, measured	44.02	1.369	1.303	1.259	—
informative, rescaled from bits/token	—	1.377	1.310	1.265	—

Two things follow.

The rescaling is exact to within half a percent. Measured bits per informative token are 1.259 for the largest model against the 1.265 the rescaling gives—a difference of half a percent. The bits-per-informative-token figures in this paper are therefore measurements, not estimates.

The inline cache is free, and the ledger says exactly how free. It is 55.56 percent of the held-out tokens and 0.25 percent of the model's held-out bits, predicted at 100.00 percent top-one accuracy at a cost of 0.0025 bits per slot. More than half the corpus is determined by a lookup table and costs a model that knows the instruction format nothing at all, with the emphasis on *nothing at all*. That also tells a reader precisely how to discount the headline: the model's 0.5569 bits per token is, to three significant figures, the cost of the 44.02 percent of the stream that carries the program, divided over the whole of it.

The ledger also isolates what the model is actually uncertain about. Opcode and argument slots are 22.01 percent of the stream each, and take 47.75 and 51.74 percent of the bits: almost exactly half each. Scale buys a little on both—top-one opcode accuracy moves from 70.11 to 72.77 percent between the smallest and largest model—and buys essentially nothing on the cache, because there was nothing there to buy.

5.3 Running the generations: the structural and execution ladder

The easiest statistic to report about these generations is that 100.0 percent of their operation bytes are legal opcodes. That number is close to vacuous, and the reason states itself: the always-legal inline-cache placeholder is a majority of the alphabet mass, so a stream can be 100 percent "valid" and still be nothing. We reproduce the statistic with the trainer's own

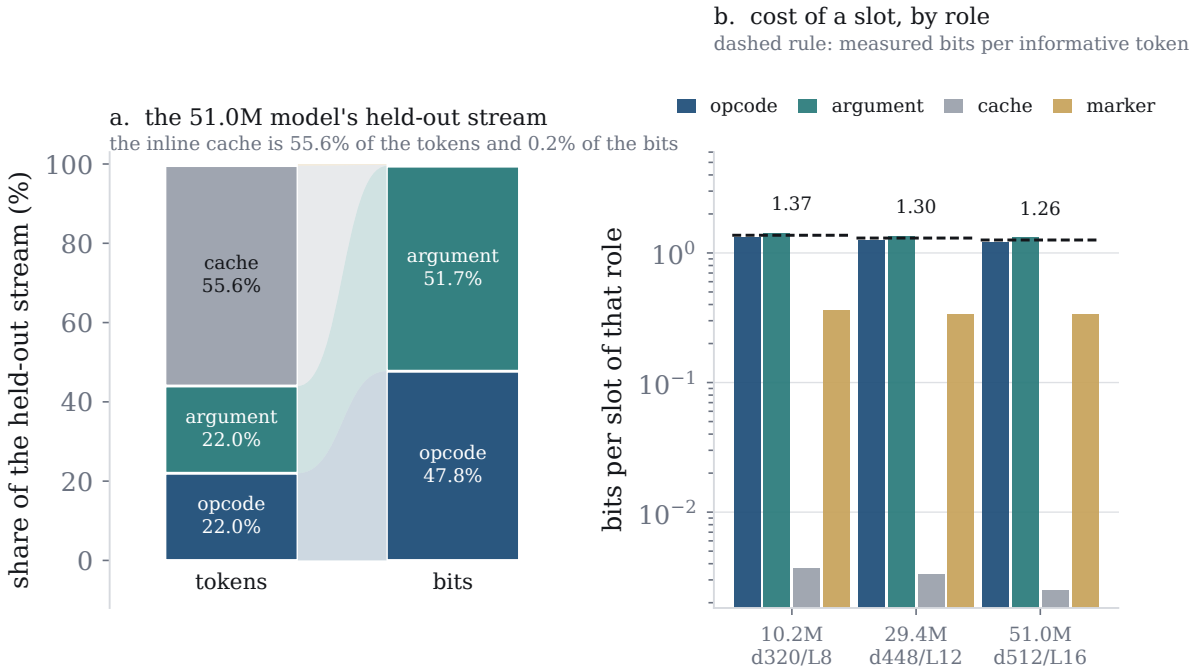

Figure 7: **Where the held-out bits actually go: token share against bit share, measured per slot role rather than assumed.** *(a)* The held-out stream of the 51.0M model by token share and by bit share. The inline-cache band is 55.6 percent of the left column and 0.2 percent of the right one; the ribbons connect the same role in both. *(b)* What a slot of each role costs, by model size, on a logarithmic axis spanning three orders of magnitude; the dashed rule is the measured bits per informative token.

procedure—greedy decoding, four hundred tokens—and confirm it: 100.0 percent of opcode slots legal, and 100.0 percent still legal with the cache placeholder excluded. Then we put that same greedy stream through a real interpreter. It disassembles, it passes the structural check, and it *segfaults CPython*.

So we ran the whole population. Five hundred generations ($T = 0.8$, cached and committed), each stripped to the byte span before its first marker symbol, wrapped in a deliberately generous `types.CodeType`—sixty-four dummy locals, sixty-five dummy constants, sixty-four dummy names, a stack size of four thousand and ninety-six, an empty exception table—and disassembled and executed in an isolated child process, because `dis.get_instructions` on malformed bytecode crashes the interpreter that calls it. A child returning -11 is a segmentation fault and is bucketed separately from a Python exception.

Table 3: The structural and execution ladder for the `co_code` model (d448/L12, 29.4M), $n = 500$ generations at $T = 0.8$, with Wilson 95% intervals. *The control column is the essential one:* it is 500 real *held-out* code objects from the same corpus, wrapped in the identical generous code object and executed identically. Real compiled code separated from its own constant and name tables also crashes, so the model's rates are only interpretable against it.

	model generations	**real held-out code (control)**
opcode bytes legal (every opcode slot)	100.0%	100.0%
excluding the cache placeholder	100.0%	100.0%
disassembles under `dis`	95.4% [93.2, 96.9]	100.0% [99.2, 100.0]
structurally valid (stack nets zero, jumps land, operands in range)	**44.2**% [39.9, 48.6]	**52.6**% [48.2, 56.9]
segmentation-faults the interpreter	53.2% [48.8, 57.5]	49.2% [44.8, 53.6]
raises a Python exception	42.0% [37.8, 46.4]	41.6% [37.4, 46.0]
runs to completion	4.8% [3.2, 7.0]	9.2% [7.0, 12.1]

Three readings, and all three hold.

First, opcode validity is not a structural metric. A metric on which the model scores 100 percent while segfaulting the interpreter on half its outputs is not measuring well-formedness, and Table 3 is what replaces it.

Second, and this one runs *in the model's favour*, the segmentation-fault rate is very largely a property of the evaluation rather than of the model. Real held-out CPython code, executed the same way, segfaults at 49.2 percent and runs to completion only 9.2 percent of the time. The mechanism is identifiable: the generous wrapper declares no arguments, so its sixty-four locals are unbound, and a bare load of an unbound local dereferences null. Under a strictly more generous wrapper that binds every local, the fault rate falls to 12.6 percent for the model against 9.2 percent for real held-out code. Separately, staging disassembly and execution in *different* child processes shows that `dis.get_instructions` never faults at all (0.0 percent of five hundred); every fault is an execution fault. A reading of "half its generations crash the interpreter" as evidence that the model produces uniquely broken artifacts is therefore wrong, and the control is what establishes that.

Third, the difference that *is* real is structural validity, and it goes against the model: 44.2 percent against the control's 52.6 percent, rising to 70.2 percent for held-out objects taken whole rather than length-matched. The model produces byte streams whose stack effects do not net to zero, whose jump targets miss instruction boundaries, or whose operands fall out of range, appreciably more often than real code does. That is the precise statement of how far the generations are from programs: not "they crash the interpreter," which real code also does here, but "a majority of them are not well-formed instruction sequences." This is learned structure, and it is not yet a program.

5.4 How much of this is retrieval?

A model trained for between two and three epochs on a 118 million-token stream invites the obvious question: are the generations *written*, or are they spans copied out of the training data? We measured, for each of the five hundred generations, the length of the longest prefix occurring verbatim at an instruction-aligned position of the training stream, with the matcher checked against a positive control (a known training substring, found in 20 of 20 attempts) and a negative control (random byte strings, longest spurious match 2 tokens).

The raw figures are large: the median generation has a 45-token verbatim prefix and 25.0 percent [21.4, 29.0] of generations occur *in their entirety* in the training corpus. Read alone, that would describe a fuzzy index over the standard library rather than a generative model.

But it cannot be read alone, and the control again matters more than the number. Real *held-out* code objects, length-matched to the model's own generations and searched the same way, are themselves 28.0 percent [24.2, 32.1] fully verbatim in the training split, with a median prefix of 37 tokens. Compiled bytecode is enormously repetitive—the same prologues, the same load-attribute-call idioms, the same returns—so long exact overlap with training is what *unseen real code* looks like too. On the headline statistic the model is therefore not distinguishable from held-out real code: the difference in fully-verbatim rate is -3.0 percentage points with a 95 percent interval of $[-8.6, 2.4]$ that contains zero. What is significant is the median prefix length, where the model exceeds the length-matched control by 8 tokens $[3.5, 10.5]$. The measurement therefore says two things: the generations copy somewhat longer spans than real unseen code does, and on the fully-verbatim measure they are not more copied than unseen real code is.

Temperature moves all of this together, and the trade-off is worth recording because it shows the metrics are not independent. Between $T = 0.6$ and $T = 1.2$ the fully-verbatim rate falls from 37.0 to 11.5 percent and the median verbatim prefix from 54 to 23 tokens—but structural validity falls with it, from 54.0 to 25.5 percent, and disassembly from 98.0 to 86.0 percent. Novelty at this scale is bought directly with well-formedness. Across this sweep there is no setting at which the model produces long, novel and well-formed bytecode at once, and that trade-off is the finding. One methodological result belongs here too: across the same sweep the run-to-completion rate moves the *wrong* way, rising as quality falls, because under this wrapper a stream that returns early without touching an unbound local counts as having run. Run-to-completion is not a quality metric on unconditional generation, and structural validity is the rung to trust.

Everything in this section is unconditional generation scored for well-formedness. The conditional, execution-graded evaluation on real held-out functions—prompt the model with a real function's tables and let it write the code section—is what measures computation, and Section 6 runs it.

6 Execution-Grounded Evaluation on Real Compiled Bytecode

The capability numbers of the previous sections are measured on one of two populations: unconditional generations scored for well-formedness, or a synthetic two-argument integer-expression grammar. The 124.2 million-token corpus of real compiled Python that the rest of the paper is about carries none of them. This section supplies the missing one. It is an execution-grounded evaluation on real compiled bytecode, it needs no new model, and it is the first result in this line of work in which a model of compiled bytecode writes code that computes the right answer on functions it has never seen.

The protocol. A held-out function is chosen, the model is prompted with the first p fraction of that function's own `co_code` cut at an instruction boundary, and the model writes the remaining bytes. The completion is spliced back into the *true* code object with `code.replace(co_code=...)`, so the constant, name, variable and exception tables are the real ones—the model is asked for the code section only, which is what a `co_code` model models—and the resulting object is executed on that function's own probe inputs in a crash-isolated child and every output compared to the true function's. This is easier than the queued full-code-object experiment in one respect, because the model is not asked to write the tables, and harder in another, because it is asked to continue real library code rather than a template.

The population, and why it is the right one. We compile every source file of the installation, extract every leaf code object that is a plausible pure function of one to four arguments, deduplicate, and then *verify*—rather than assume—that the object does not occur in the training split every reported checkpoint was trained on. That test is exact: the corpus is `co_code`

delimited by markers, so splitting the training prefix on those markers recovers the training code objects themselves. Of 102,270 deduplicated candidates, 83.75 percent are training objects and 16,619 are not. Each survivor is then required to admit at least eight probe input tuples on which the true function returns deterministically, with at least *two distinct outputs* among them, so that a generation which merely returns a constant cannot pass; the probe search is deterministic given the function and a fixed seed, and is re-derived inside the grading child rather than shipped to it. That leaves 1,293 functions, of which none occurs even as a verbatim *substring* of the training stream. The pre-registered criterion asked for at least three hundred.

The controls, which matter more than the number. Six completions are graded for every function and every prompt length. `true` is the real continuation and is a harness self-test: it must score 100 percent exact-output agreement, and the script refuses to write a receipt if it does not. `randinit` is an identically-shaped transformer with random weights. `trigram` is an opcode-trigram plus $P(\text{argument} \mid \text{opcode})$ sampler fitted on the model's own training split—a learned-marginals control. `donor` is the true continuation of a *different* function of sufficient length. `filler` is a run of the inline-cache byte. The last two exist because of a confound that turns out to be the dominant one in this experiment: a prompt that already contains the function's return path decides the outputs by itself, and a completion carrying no information at all will then be scored correct.

What was measured. Four hundred and fifty of the 1,293 functions were graded, a seeded random subsample chosen for what a shared CPU could complete. Twenty-four are dropped because their probe set could not be re-derived deterministically inside the grading child, and one more—`signal.signal`, which installs a handler and therefore returns something different once it has been called—is dropped because the *true* continuation itself fails on it. That exclusion looks only at the `true` arm, never at the model's, and it is applied identically to every arm. On the remaining 425 functions the harness self-test passes: the true continuation reproduces every probe output of every function at every prompt length.

Table 4: Exact-output agreement with the true held-out function: every one of a median of 16 probe inputs must return exactly what the real function returns. *Left block:* all 425 functions. *Right block:* the decisive subpopulation—functions on which neither degenerate control reproduces the true outputs, so the completion is load-bearing—defined by the controls and never by the model. Wilson 95% intervals.

	all 425 functions			**decisive subpopulation**		
prompt fraction	0.25	0.50	0.75	0.25	0.50	0.75
(objects)				412	401	384
`true` (self-test)	100.0	100.0	100.0	100.0	100.0	100.0
model (greedy)	2.12	3.76	**11.76**	1.21	2.00	**7.55**
Wilson 95%	[1.12,3.98]	[2.33,6.03]	[9.04,15.18]	[0.52,2.81]	[1.01,3.89]	[5.31,10.64]
`filler` (all-cache)	3.06	5.65	9.65	0.0	0.0	0.0
`donor` (another function)	0.47	3.06	5.88	0.0	0.0	0.0
`trigram`	0.0	0.0	0.71	0.0	0.0	0.0
`randinit`	—	0.24	—	—	0.0	—

The result. Given three quarters of a real held-out library function, the model writes a completion that reproduces *every* output of that function on 11.76 percent [9.04, 15.18] of them. On the decisive subpopulation—where a completion carrying no information cannot succeed—it reaches 7.55 percent [5.31, 10.64] while *every* control scores exactly 0.0 percent [0.00, 0.99]. This

is the first number in this line of work that says a model of compiled bytecode wrote something that computed the right answer on code it had never seen, and it is not explained by copying: only 4.71 percent of completions are byte-identical to the truth, and among those that are not, 8.96 percent [6.34, 12.50] are still output-exact.

The pre-registered criterion, and the verdict on it. The criterion was registered before the run: "at least ten percent exact-output agreement on at least three hundred held-out functions, with a Wilson interval excluding zero." On its literal terms, at a three-quarter prompt, it is *met*: 11.76 percent on 425 functions with an interval excluding zero. The same table also shows an all-cache filler scoring 9.65 percent on that population, with a paired test that cannot separate the model from it ($p = 0.25$, 29 discordant pairs in the model's favour against 20 against), which is the signature of the long-prompt confound: at three quarters, a prompt often decides a function's output by itself. On the decisive subpopulation, where that confound is removed, the model scores 7.55 percent against a criterion of ten, and that is the number the controls make load-bearing.

Against the learned-marginals null, the model wins decisively. The opcode trigram, which knows the instruction format and third-order opcode statistics, reaches 0.71 percent at the longest prompt and 0.0 percent on the decisive subpopulation. Paired McNemar tests give $p < 0.001$ at every prompt length (16 to 48 discordant pairs, at most one against the model). A random-init transformer of identical shape reaches 0.24 percent. Whatever the model has, it is not the marginals.

The rungs below the top one. Only 33.88 percent [29.54, 38.51] of the model's three-quarter-prompt completions disassemble at all, against 100.0 percent for the real continuation, and 27.29 percent [23.28, 31.72] run to completion. The model is therefore not producing well-formed instruction sequences that happen to be wrong; it is producing malformed ones most of the time, and the successes are a minority tail. The trend across prompt length is monotone and steep—2.12, 3.76, 11.76 percent—so the summary the measurements support is that this model finishes a function it has been shown most of, and does not write one.

6.1 How far greedy decoding carries a real program

Before asking whether a completion *computes* the right thing, it is worth asking the simpler question: how much of a real function can the model reproduce at all? For a real code object, teacher-forced greedy prediction and free-running greedy decoding are identical up to and including the first disagreement, so the index of that first disagreement is exact from a single forward pass, with no sampling and no variance. We measure it on all 1,293 held-out objects of the evaluation population.

The measurement locates the ceiling on unrolled generation exactly. Under greedy decoding the flagship 29.4M model reproduces a median of 2 instructions of a real held-out function before its first error, a mean of 2.27 [2.24, 2.30]; the 51.0M model reaches 2.67 [2.62, 2.72] and the 10.2M model 2.59 [2.54, 2.63], so the ordering is not even monotone in size. *Not one* of the 1,293 objects is reproduced in its entirety (0.0 percent, Wilson [0.00, 0.30]). An opcode trigram fitted on the same training split survives 2.52 [2.48, 2.56] instructions—*longer* than the flagship model—while a random-init transformer of identical shape survives 1.00 [1.00, 1.00], which is the floor.

The per-token picture is different, and the gap between the two pictures is the finding. Measured as teacher-forced top-one accuracy over the object's bytes, the flagship model is right on 85.00 percent of tokens against the trigram's 72.26 percent and random init's 0.31 percent. The model is therefore substantially the better one-step predictor and barely the better unrolled

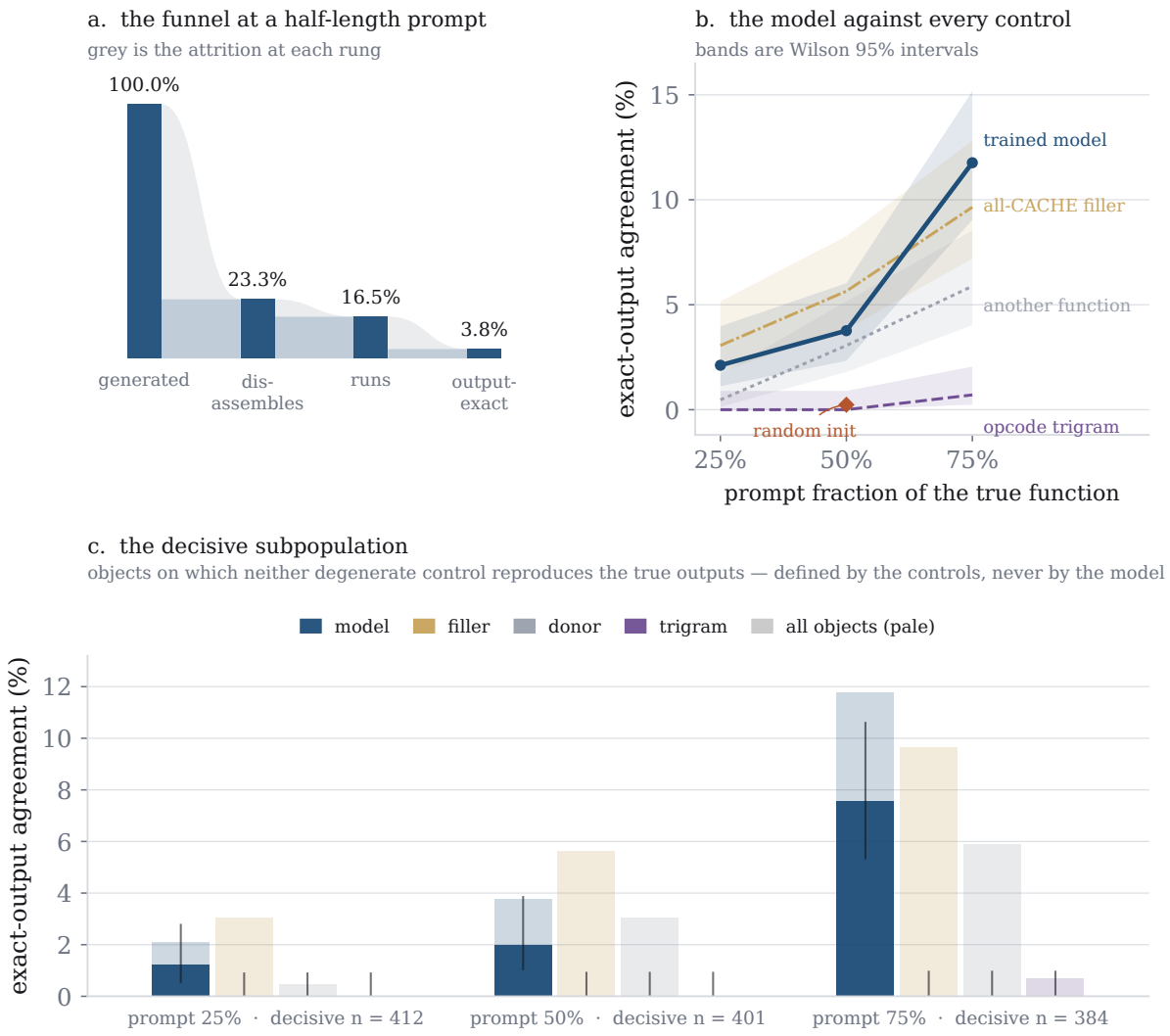

Figure 8: **The model completes a held-out library function, and the completion is run against the true function.** Execution-grounded evaluation on real compiled bytecode: $n = 425$ stdlib code objects, each verified byte-absent from the training stream and admitting at least eight deterministic probe inputs, where *exact* means every probe output equals the true function's. *(a)* The ladder at a half-length prompt: of the model's completions, 23.29% disassemble, 16.47% run, and 3.76% reproduce every output of the true function; grey is the attrition at each rung. *(b)* Exact-output agreement against every control, with Wilson 95% bands. *(c)* The same quantity on all objects (pale) and on the decisive subpopulation (solid), where every degenerate control is exactly zero and the model is not.

generator, because errors compound: at 85.00 percent per token, the expected run before a mistake is short no matter how much better than a trigram that eighty-five percent is. One-step compression and unrolled generation are therefore separate capabilities on this substrate, and this paragraph measures the second.

Where the first mistake falls is informative about what has and has not been learned. Of the first errors, 82.13 percent land on an opcode slot and 17.87 percent on an argument slot; 0.00 percent land on an inline-cache slot or a marker. The model has the instruction *format* essentially perfectly—it never mispredicts the deterministic part—and fails on the choice of the next operation, which is the part that carries the program.

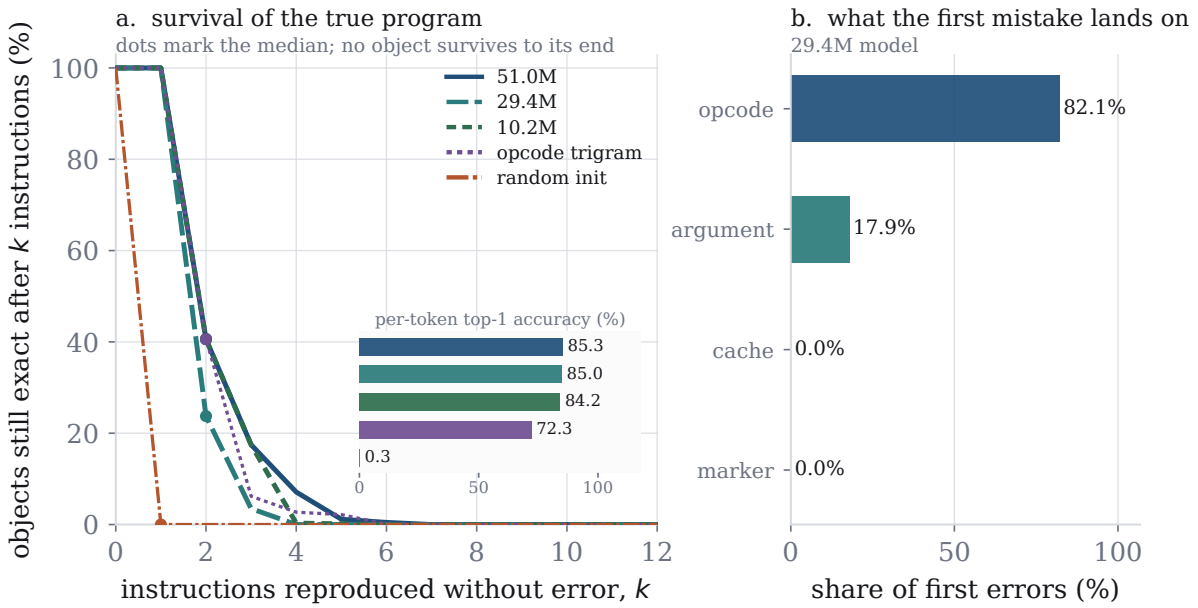

Figure 9: **How far greedy decoding survives on real code: instructions reproduced before the first disagreement with the true function.** Teacher-forced and free-running greedy decoding agree exactly up to the first divergence, so the index is exact from one forward pass, with no sampling and no variance. *(a)* The fraction of real held-out code objects whose greedy continuation is still byte-exact after k instructions, for the three model sizes against an opcode-trigram and a random-init control, $n = 1{,}293$ objects verified byte-absent from the training split. Dots mark the median; the inset bars are per-token top-one accuracy. The curves fall to near zero within a handful of instructions and no object survives to its end. *(b)* What the first mistake lands on, for the 29.4M model: never the format-determined slots, almost always the opcode.

7 Is the Substrate CPython, or Compiled Code?

One open question stands above the others in this line of work: whether the tractability reported here survives an instruction set without CPython's inline caches. This section answers it at the level of the substrate, and the answer is that the regularity is a property of compiled code and not of CPython.

We rebuild the corpus statistics of Section 5.1 on three further byte streams, measured with identical estimators on identically constructed samples—sixty-four contiguous blocks spread evenly over each stream, rather than a prefix, so that no substrate is judged on its first few files. The streams are: the `.text` sections of sixty ELF binaries taken from this machine's `/usr/bin` and `/usr/lib/x86_64-linux-gnu`, disassembled with `objdump` into 19,891,694 x86-64 instructions over 79.12 megabytes; the code section of a WebAssembly module, decoded by a self-checking decoder that is required to land exactly on each function body's final byte and does so for all 1,638 bodies, giving 306,498 instructions; and, as the reference point the motivating argument invites, 24.01 megabytes of the *source text* of the same installation. Alongside them we place CPython's stream both as the model sees it and with the deterministic inline-cache slots removed.

Table 5: The same statistics on four instruction substrates and one source-text control, measured identically. $H(o' \mid o)$ is Eq. (3). *The column that answers the paper's open question is $H(o' \mid o)$:* CPython and x86-64 agree to within two hundredths of a bit, although x86-64 has no inline-cache slot at all, an opcode alphabet two and a half times larger in the same sample, a variable-length encoding, and comes from C and C++ compilers rather than from CPython's. The column that complicates the story is the last one.

	instructions	$\lvert\mathcal{O}\rvert$	$H(o)$ bits	$H(o' \mid o)$ bits	$H(o' \mid o, o'')$ bits	det. slots % of stream	`xz` bits/insn
CPython 3.12	26,596,169	103	4.747	3.114	2.379	57.0	4.777
cache-free	26,596,169	103	4.747	3.114	2.379	0.0	4.069
x86-64	19,891,694	268	4.116	**3.096**	2.698	0.0	10.292
WebAssembly	306,498	111	3.925	2.477	1.994	0.0	2.134
Python source text	—	—	—	—	—	0.0	—

An instruction is not a defined unit of source text, which is why the last row of Table 5 is empty in the instruction-level columns. The *byte* is a defined unit of all five streams, so Table 6 states the same comparison in that unit, where the source-text control carries numbers rather than dashes and the two units can be read against each other.

Table 6: The same five streams in the one unit they all share, the byte. $H(b)$, $H(b' \mid b)$ and $H(b' \mid b, b'')$ are byte entropies on identically constructed samples; the last three columns are what each classical compressor spends on a byte. Read against Table 5: CPython's raw stream is the cheapest per byte of any of them, and it is the inline-cache filler that makes it so—with the cache slots removed the same code costs 2.03 bits per byte, more than its own source text at 1.40.

	megabytes	$\lvert\Sigma\rvert$	$H(b)$ bits	$H(b' \mid b)$ bits	$H(b' \mid b, b'')$ bits	`gzip`	`bzip2` bits/byte	`xz`
CPython 3.12	123.72	256	3.068	2.324	1.676	1.539	1.133	**1.027**
cache-free	53.19	256	5.421	4.174	2.977	2.984	2.529	2.034
x86-64	79.12	256	6.233	4.506	3.168	3.309	3.052	2.587
WebAssembly	0.72	253	5.219	3.423	2.427	2.399	1.262	0.908
Python source text	24.01	208	4.768	3.676	2.720	1.782	1.599	1.404

Three readings, and all three hold.

The grammar transfers. The quantity the paper's mechanism argument rests on—the entropy of an operation given its predecessor—is 3.114 bits on CPython and 3.096 bits on x86-64. Those are not similar numbers by coincidence of scale; they are the same number. Whatever makes a compiled operation stream predictable one step at a time is therefore not the inline-cache encoding, is not the small alphabet (x86-64's is larger), and is not CPython's compiler. WebAssembly, a third stack machine from a third toolchain, is lower still at 2.477 bits. The near-deterministic successor structure that a small model exploits is therefore a property of compiled code.

The density does not transfer, and this is the complication. Measured in bits per *instruction*—the only unit in which instruction sets of different densities can be compared—`xz` spends 4.777 bits on a CPython instruction, 2.134 on a WebAssembly instruction, and 10.292 on an x86-64 instruction. An x86-64 instruction carries registers, addressing modes, displacements

and immediates that a CPython instruction delegates to a table, so it is more than twice as expensive even though its opcode is no less predictable. Table 5 is therefore a statement about structure rather than about density: the regularity a model trained on x86-64 would have to learn is there, and the cost of encoding one of its instructions is a separate quantity.

Compiled is not simply "smaller" than source. On identical estimators, `xz` attains 1.03 bits per byte on CPython's stream and 1.40 on the source text of the same installation—but that comparison is won by the inline-cache filler. With the cache slots removed, the compiled stream costs 2.03 bits per byte, above its own source text. What the compiled substrate offers is therefore *learnability* rather than density: the alphabet is finite and small and its successor distribution is sharp.

The scope of this section is the substrate itself. No model is trained on any of these streams; the matched-capacity, matched-budget transformers that carry the result to the model level—together with an instruction-level shuffle control that separates grammar from marginals on every substrate at once—are specified with pre-registered criteria and queued. The WebAssembly corpus is a single module from a single toolchain, so its numbers describe one compiler's output and stand as a third data point on the instruction set. Of the x86-64 instructions, 1,319,479—6.63 percent—are `objdump`'s `(bad)` decodings of data interleaved in `.text`; they are retained in the byte stream and counted, because discarding them would clean the substrate we are claiming is tractable.

8 Beyond Compression: Probing the Frozen Model

The sharpest structural question is whether the agreement statistics can be matched, or beaten, by a sampler that knows only the corpus's opcode marginals—a point Section 5.1 quantifies. A compression number cannot settle it. A probe can: we freeze the trained model, read out its final-layer representation of a real code object, and ask a linear classifier to recover properties of the program that the next-byte objective was never asked to predict.

The design is built around its nulls, because the absolute accuracy of a probe means very little. We report three featurisers of the identical objects: the trained model, an *identically shaped transformer with random weights* (a randomly initialised transformer is already a strong featuriser, so the trained-minus-random gap is the quantity that carries meaning), and an explicit *bag-of-opcodes* histogram, which is the marginals-only null made concrete. Cross-validation is grouped by source file, so two functions from the same module never straddle a fold; without that, package prediction leaks. Labels come from CPython itself or from executing the true function, never from us.

Table 7: Linear probes on frozen representations of real held-out code objects, five-fold cross-validated with folds grouped by source file, accuracy with a bootstrap 95% interval over folds. *The comparison is horizontal, not vertical:* what matters is whether the trained model beats a random-init transformer of the same shape *and* an explicit marginals-only featuriser. It does so on one task of four, decisively; on one it is ambiguous; on two the marginals win.

task	what it asks	majority	**trained**	random init	bag of opcodes
`argcount`	how many arguments	38.21	**70.77**	60.10	52.14
`rettype`	what type it *returns*	29.02	**54.02**	45.38	51.71
`has_loop`	does it contain a backward jump	76.64	96.83	96.60	**99.00**
`package`	which library it came from	36.20	39.90	35.33	**41.29**

One task the representation clearly carries. On `argcount` the trained model reaches 70.77 percent [66.62, 74.84] against 60.10 percent [56.12, 63.75] for the random-init transformer

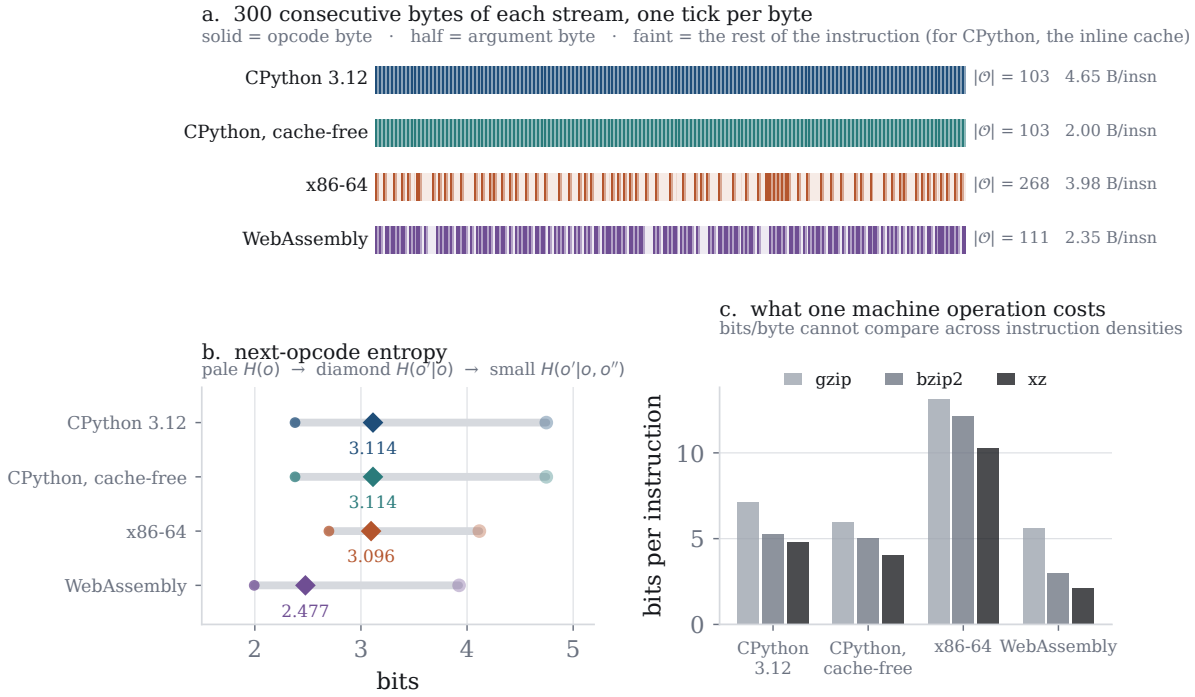

Figure 10: **Is the regularity a property of CPython, or of compiled code?** The same statistics on four instruction substrates, measured with identical estimators; no model is trained here. *(a)* Three hundred consecutive bytes of each stream, one tick per byte, with a solid tick for an opcode byte, a half-height tick for an argument byte and a faint tick for the rest of the instruction, which for CPython is the inline cache. *(b)* The entropy of an opcode unconditionally, given its predecessor, and given its two predecessors; the diamond is $H(o' \mid o)$, Eq. (3). CPython and x86-64 coincide at the diamond. *(c)* What one machine operation costs the classical compressors, the unit in which instruction sets of different densities can be compared; here the substrates separate sharply and x86-64 is the most expensive.

and 52.14 percent $[46.41, 57.51]$ for the opcode histogram, on a four-way problem whose majority class is 38.21 percent. The same ordering holds on the larger structural population of 1,800 objects (69.39 against 60.33 and 55.89). Nothing in the training objective asked the model to represent a calling convention; a linear read-out of its final layer recovers one, and does so well beyond what the opcode inventory supports.

One behavioural task where it is ahead on accuracy and behind on balance. The `rettype` probe is the one that is genuinely about *semantics*: the label is the type the true function actually returns when executed on its own probe inputs, and nothing about a return type is present in `co_code`. The trained model reaches 54.02 percent $[50.20, 58.69]$ on an eight-way problem against 45.38 percent for random init and 51.71 percent for the opcode histogram. The margin over the marginals null is 2.3 points, well inside that interval, and on macro-F_1 the ordering *reverses*: 46.83 for the trained model against 48.59 for the bag of opcodes. The representation therefore encodes something about what the code returns, and the opcode inventory alone supplies most of it.

Two tasks the marginals win, on a suite fixed before it was run. On `has_loop` the bag of opcodes reaches 99.00 percent, above the trained model's 96.83—as it should, because a backward jump *is* an opcode, and a task that lives in the marginals ought to be solved by the marginals. That result is the suite's own sanity check. On `package` no featuriser separates from the others; the trained model's 39.90 percent [32.71, 46.49] overlaps the histogram's 41.29 percent. There is no evidence here that the model has learned library-specific idiom beyond opcode frequency.

Table 7 separates two hypotheses that are usually conflated. The representation is not *only* marginals—the argument-count result is far outside what an opcode histogram supports, and outside a random-init transformer of the same shape—while on the one task that is genuinely about behaviour rather than structure the margin over the marginals null is the two-point one measured above. That is a sharper answer than either "it has learned the grammar" or "it has learned the marginals."

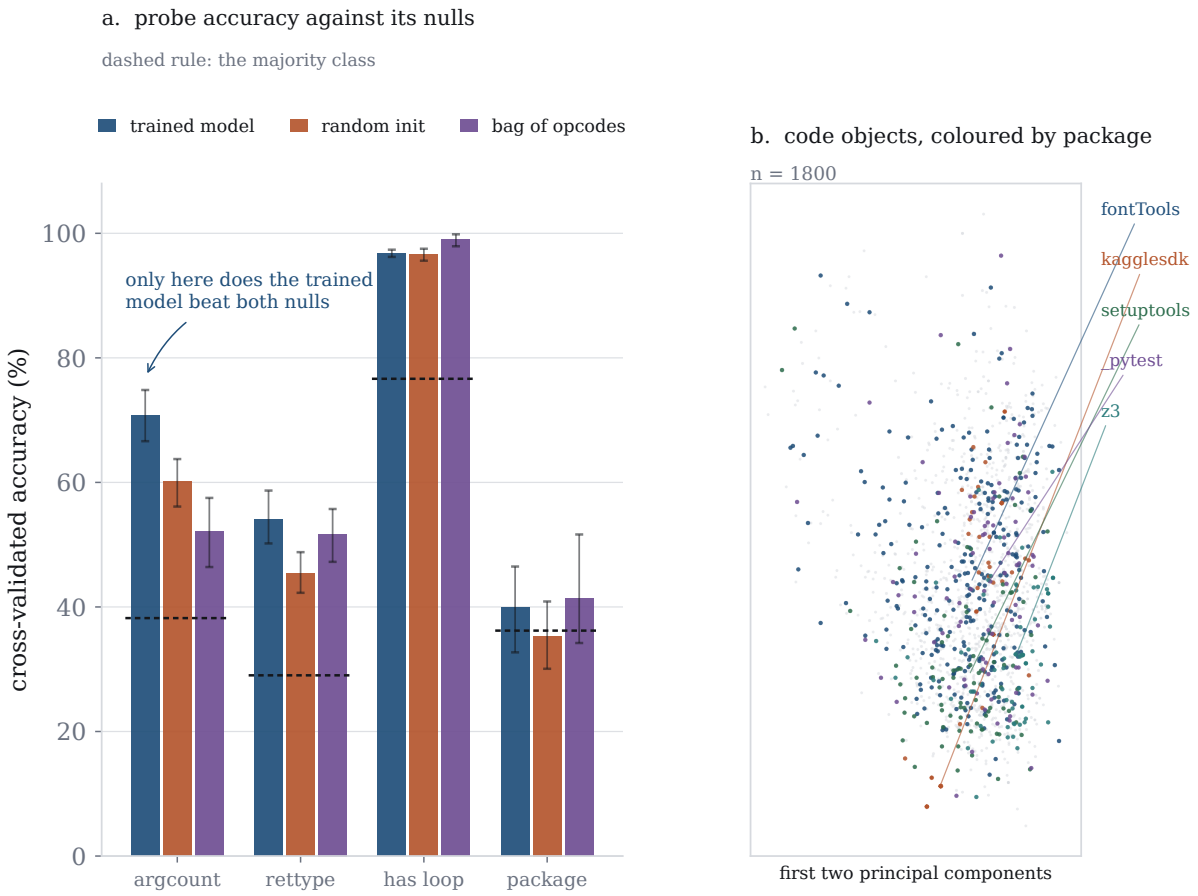

Figure 11: **A linear probe on frozen representations: what a read-out of the model recovers that the next-byte objective never asked for.** The quantity that carries meaning is the gap to a random-init transformer of the same shape and to an explicit marginals-only featuriser, not the absolute accuracy. *(a)* Cross-validated probe accuracy for each task and featuriser, with bootstrap 95% intervals over folds and the majority class as a dashed rule; folds are grouped by source file, so two functions from the same module never straddle a fold. The trained model separates from both nulls on `argcount` only. *(b)* The first two principal components of the frozen representation of 1,800 held-out code objects, coloured by the package they came from; the weak separation here is the same finding as the `package` row of Table 7, shown rather than asserted.

9 Connection to the General Purpose Neural Computer

A finite operation alphabet is precisely the primitive vocabulary that a neural computer requires. In the companion work on the general purpose neural computer, the *semantics* of such an alphabet are learned from the observation of execution; here the *distribution* over operation sequences that real programs occupy is learned from a corpus of compiled code. The union of the two is a neural system that both writes operation sequences and knows what those sequences do—a system that writes and runs real, auditable programs.

10 From Structural to Runnable Generation

The generations described so far are structurally plausible but not runnable: the raw operation bytes, spliced into a code object and executed, fail, because their arguments index the constant and name tables of the code object, which a model of `co_code` alone does not co-generate. The natural response is to stop discarding that metadata. We serialize the *entire* code object—operation stream together with its constant table, name table, argument counts, and stack size—into a single byte-token stream, framed by a small set of structural marker tokens (raising the vocabulary from two hundred and fifty-eight to two hundred and seventy-eight), with integer constants encoded as sign-and-varint byte sequences. A model trained on this representation emits, for each generation, both the operations and the tables they reference.

To measure the result we built a verification harness that reconstructs a `types.CodeType` from a generation and executes it in an isolated subprocess under strict resource limits—bounded processor time and memory, no file writes, and an emptied builtin namespace so that no external primitive is reachable—so that a malformed or hostile generation is contained to a disposable child rather than trusted, together with an oracle that symbolically decompiles the generated operations to an arithmetic expression and checks that the virtual machine's output equals the expression's value on a battery of eight input pairs. The oracle scores the ground-truth corpus at 150 of 150. On a corpus of two rungs of the difficulty ladder at once—straight-line integer functions and, added on top, functions with a conditional branch that the oracle checks as a ternary—a fourteen-million-parameter model trained on this serialization reaches a runnable rate of 99.4 percent and a correct rate of 99.2 percent $[97.96, 99.69]$ over five hundred sampled generations.

Four properties of that result are measured rather than asserted, and two of them fix its scope.

The oracle is self-consistency, not correctness. `decompile_expr` re-implements LOAD_FAST, LOAD_CONST, BINARY_OP, COMPARE_OP and RETURN_VALUE using Python's own operators, in the same interpreter that then executes the bytecode, so agreement with CPython is close to tautological. It establishes that the executed bytecode computes what its own disassembly says, which is a real property, and it is not correctness against an external specification, because there is no statement of what these programs *should* compute. We now route the decompiled expression back through CPython's own `compile()` as a fresh function, and execute *that* on sixty-four seeded probes including negatives: an independent compilation path and an eightfold wider probe set. The eight shipped probes contain no negative operand at all, while the generator draws its arguments from $[-9, 12]$ and its literals from $[-20, 20]$, so this was a live worry. It turns out not to matter: the sixty-four-probe correct rate is 99.2 percent, identical to the eight-probe rate, with zero discordant pairs and a McNemar p of 1.0. The fixed probe set is not a confound. The oracle is not independent of the interpreter it checks, and self-consistency is what it delivers.

The "opaque" bucket. A generation using any opcode outside the modelled six-opcode subset cannot be decided by the decompiler and is counted as *incorrect*, which means the correct rate also measures "stayed inside the template." Broken out, the opaque rate is 0.0 percent $[0.00, 0.76]$: on this corpus the model never leaves the template. That is a clean result for the correctness number and a narrow one for the capability.

The harness repairs the model's output, and the repair is not load-bearing. The reconstruction over-provisions the frame with `max(stacksize, 2048)`, so the model's own generated `co_stacksize`—minimum 0, median 4, maximum 5—is discarded on *every single generation*, and a runnable rate measured through it would describe the harness as much as the generation. A strict mode keeps the model's number verbatim, executing in the crash-isolated child (a starved frame aborts the child at around twenty thousand stack slots and segfaults it at around one hundred thousand; the smoke test asserts this before the run). Strict runnable is 99.4 percent $[98.25, 99.80]$—identical to lenient—with a strict segmentation-fault rate of 0.0 percent. The model emits the exactly-required stack size on 99.4 percent of generations and under-provisions on 0.4 percent. Removing the repair therefore does not move the number. CPython 3.12 carves frames from a pre-allocated datastack chunk, so part of the low strict fault rate is slack absorbing small errors rather than the generated stack size being right.

Novelty is a deficit against a ceiling, not an absolute rate. Novelty here means "this signature did not occur in training," and measuring it takes two things. The first is a keyset covering every training program: built from 80,000 programs while the model was trained on 100,000, it would score 15,368 of the 20,000 missing programs—76.8 percent of them—as *novel* even had the model regurgitated them verbatim. Completing the keyset moves the rate only from 73.4 to 72.8 percent, which is itself informative. The second is a ceiling, and it is the one that decides the reading: a sampler with *zero* memorisation by construction does not score one hundred percent, because the training corpus collides with itself 22.2 percent of the time. Drawing two thousand fresh programs from the paper's own generator—the data distribution itself, which cannot memorise—gives a novelty ceiling of 78.8 percent $[76.95, 80.54]$. The model scores 72.8 percent $[68.74, 76.52]$, a deficit of 6.0 percentage points with a 95 percent interval of $[-10.42, -1.85]$ that excludes zero. On the stricter structural signature, with constants abstracted away, the deficit is 6.35 points. **The model is significantly *less* novel than fresh draws from its own data distribution.** It recombines the template at roughly ninety-two percent of the rate the generator itself does, and the novel-correct rate is 72.4 percent $[68.32, 76.14]$ read against that ceiling rather than against zero.

Because the oracle is cheap, a single generation is not the operative quantity: one may sample repeatedly and keep a verified program. Generation here is unconditional (there is no posed problem), so the quantity is the reliable *yield* of valid, template-conforming programs under repeated sampling rather than problem-solving `pass@`k. Every number in this section is measured on a synthetic two-argument integer-expression grammar; Section 6 is the execution-grounded result on the 124.2 million-token corpus of compiled Python.

11 Open Problems

The runnable result of Section 10 now covers straight-line integer functions and a first control-flow rung, the conditional branch; the agenda is to keep climbing the difficulty ladder one virtual-machine feature at a time—names and global lookups over a whitelist, loops with their exception tables, nested code objects, and finally real library functions—holding novel-correct as the metric at each rung. The first and most important rung is not on that ladder at all but beside it: the same evaluation on *real* compiled bytecode rather than on a synthetic grammar. Section 6 takes the first step on it, grading completions of real held-out functions by execution.

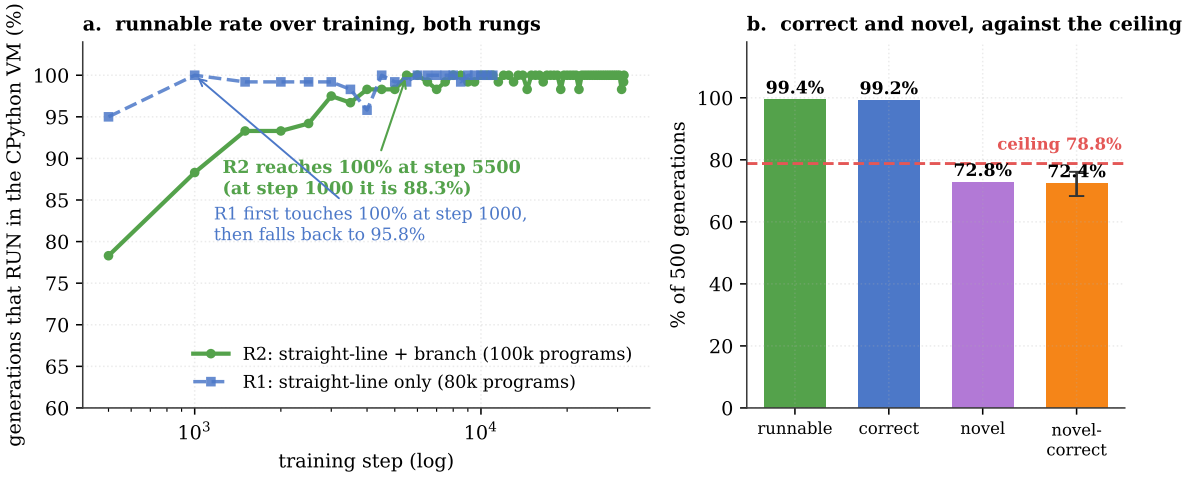

Figure 12: **Runnable bytecode generation on the synthetic grammar.** *(a)* The fraction of generations that reconstruct into a real code object and execute in the CPython virtual machine, for both rungs: R1 is straight-line integer functions and R2 adds a conditional branch. R2 first reaches 100% at step 5500 and is at 88.3% by step 1000, the step at which R1 first touches 100% before falling back below it. Each curve point is 120 generations, so single points are noisy. *(b)* Over 500 generations, 99.2% are correct and 72.8% novel, giving novel-correct 72.4% (Wilson [68.32, 76.14]). The dashed rule is the control that makes novelty readable: fresh draws from the data generator itself, which cannot memorise, are 78.8% novel [76.95, 80.54]—*above* the model—so novelty is a deficit against that ceiling rather than an absolute rate. The oracle scores the ground-truth corpus at 150/150.

The step that remains is the full code object: the corpus is built—leaf, self-contained code objects from the standard library, held out by top-level package, with constants, names and exception tables serialized alongside the operations—and the training is queued behind a pre-registered criterion of at least ten percent exact-output agreement with the true held-out function on at least eight probe inputs, with a Wilson interval excluding zero, published whichever way it falls.

Beyond it lie the learning of a translation from source to bytecode, which is a learned compiler, and the prediction of behaviour from bytecode, which is differentiable execution.

The extension of the substrate beyond one virtual machine is no longer a conjecture at the level of the substrate. Section 7 measures it: with identical estimators the next-opcode conditional entropy is 3.114 bits on CPython's stream, 3.096 bits on x86-64 machine code, which has no inline caches at all, and 2.477 bits on a WebAssembly code section, so the regularity a small model exploits is a property of compiled code. The sharper question that replaces it is whether a *model* trained on those substrates reaches comparable bits per instruction, since the cost of encoding one operation differs across them by a factor of two. That experiment is specified with pre-registered criteria and queued.

12 Conclusion

Compiled binary is a viable and efficient substrate for neural modeling. A small model learns much of the statistical structure of a real virtual machine's bytecode directly, from bytes alone and with no natural language, and beats every classical compressor we could put against it on the same held-out bytes—by a factor of about two, measured with the last-twenty estimator and against the strongest of those compressors.

That headline is reported in the unit that separates the program from the format: bits per

informative token, measured slot by slot with a forward pass rather than rescaled from a total.

Three results carry the argument. The first is that the substrate's regularity is not a CPython artifact: an instruction set with no inline caches, a larger alphabet and a different compiler has the same next-opcode conditional entropy to within two hundredths of a bit. The second is that the model has, for the first time, been shown to write compiled code that *computes the right answer* on functions it has never seen—at a rate of a few percent, on a subpopulation where every control scores zero, and only when most of the function is given to it. The third names the next problem: unrolled greedy decoding leaves the true program after two instructions, and an opcode trigram does no worse. A model that is an excellent one-step predictor of compiled code and a developing generator of it is exactly what these three results describe together, and closing that gap—not further compression—is the problem this line of work now faces.

What survives is the direction. Coupled with a neural computer that grounds the *meaning* of the operations, the direct modeling of compiled binary points toward neural systems that operate upon the true substrate of computation rather than upon its human-facing shadow—and, unlike source text, that substrate can be executed, so every claim about it is falsifiable by running it. That is the property we intend to exploit next.

A Reproducibility

Every measurement is produced by an experiment script that writes a machine-readable receipt: the value, the sample size, the confidence interval, the input files, the checkpoint, the seed, the git revision, the host and the timestamp. Figure scripts *read* those receipts and fail loudly if one is absent rather than falling back to a literal, so no figure can carry a number that no measurement produced. A verification pass extracts every numeral in this document, re-derives it from its receipt, and exits with an error on any numeral that has drifted or that has no receipt and is not on a short, individually justified list of definitional constants; the same pass runs a set of executable guards, each pinning a measurement to the sentence it licenses. All generations analysed here are a single identified population, cached and committed, rather than a fresh sample drawn by each figure.

References

[1] J. L. Pharr. *The General Purpose Neural Computer.* Georgia Cyber Warfare Range, 2026.

[2] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages.* Findings of EMNLP, 2020. arXiv:2002.08155.

[3] Y. Wang, W. Wang, S. Joty, S. C. H. Hoi. *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation.* EMNLP, 2021. arXiv:2109.00859.

[4] M. Chen et al. *Evaluating Large Language Models Trained on Code.* 2021. arXiv:2107.03374.

[5] Y. Li et al. *Competition-level Code Generation with AlphaCode.* Science 378(6624):1092–1097, 2022.

[6] R. Li et al. *StarCoder: May the Source Be With You!* 2023. arXiv:2305.06161.

[7] B. Rozière et al. *Code Llama: Open Foundation Models for Code.* 2023. arXiv:2308.12950.

[8] B. Rozière, M.-A. Lachaux, L. Chanussot, G. Lample. *Unsupervised Translation of Programming Languages.* NeurIPS, 2020. arXiv:2006.03511.

[9] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, D. Song. *Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection.* ACM CCS, 2017.

[10] X. Li, Y. Qu, H. Yin. *PalmTree: Learning an Assembly Language Model for Instruction Embedding.* ACM CCS, 2021. arXiv:2103.03809.

[11] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, C. Zhang. *jTrans: Jump-Aware Transformer for Binary Code Similarity Detection.* ISSTA, 2022. arXiv:2205.12713.

[12] N. Jiang, C. Wang, K. Liu, X. Xu, L. Tan, X. Zhang, P. Babkin. *Nova: Generative Language Models for Assembly Code with Hierarchical Attention and Contrastive Learning.* ICLR, 2025. arXiv:2311.13721.

[13] C. Fu, H. Chen, H. Liu, X. Chen, Y. Tian, F. Koushanfar, J. Zhao. *Coda: An End-to-End Neural Program Decompiler.* NeurIPS, 2019.

[14] H. Tan, Q. Luo, J. Li, Y. Zhang. *LLM4Decompile: Decompiling Binary Code with Large Language Models.* EMNLP, 2024. arXiv:2403.05286.

[15] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, C. K. Nicholas. *Malware Detection by Eating a Whole EXE.* AAAI Workshops, 2018. arXiv:1710.09435.

[16] W. Zaremba, I. Sutskever. *Learning to Execute.* 2014. arXiv:1410.4615.

[17] S. Reed, N. de Freitas. *Neural Programmer-Interpreters.* ICLR, 2016. arXiv:1511.06279.

[18] P. Veličković, R. Ying, M. Padovano, R. Hadsell, C. Blundell. *Neural Execution of Graph Algorithms.* ICLR, 2020. arXiv:1910.10593.

[19] Y. Ding, B. Steenhoek, K. Pei, G. Kaiser, W. Le, B. Ray. *TRACED: Execution-aware Pre-training for Source Code.* ICSE, 2024. arXiv:2306.07487.

[20] C. Liu, S. Lu, W. Chen, D. Jiang, A. Svyatkovskiy, S. Fu, N. Sundaresan, N. Duan. *Code Execution with Pre-trained Language Models.* Findings of ACL, 2023. arXiv:2305.05383.

[21] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, D. Tarlow. *DeepCoder: Learning to Write Programs.* ICLR, 2017.

[22] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, P. Kohli. *RobustFill: Neural Program Learning under Noisy I/O.* ICML, 2017. arXiv:1703.07469.

[23] X. Chen, C. Liu, D. Song. *Execution-Guided Neural Program Synthesis.* ICLR, 2019.

[24] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, S. C. H. Hoi. *CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning.* NeurIPS, 2022. arXiv:2207.01780.

[25] X. Chen, M. Lin, N. Schärli, D. Zhou. *Teaching Large Language Models to Self-Debug.* ICLR, 2024. arXiv:2304.05128.

[26] G. Delétang et al. *Language Modeling Is Compression.* ICLR, 2024. arXiv:2309.10668.

[27] F. Bellard. *NNCP v2: Lossless Data Compression with Transformer.* Technical report, bellard.org, 2021.

[28] M. Hutter. *The Hutter Prize for Lossless Compression of Human Knowledge.* 2006. http://prize.hutter1.net/.

[29] R. Eldan, Y. Li. *TinyStories: How Small Can Language Models Be and Still Speak Coherent English?* 2023. arXiv:2305.07759.

[30] S. Gunasekar et al. *Textbooks Are All You Need.* 2023. arXiv:2306.11644.

[31] L. Ben Allal et al. *SmolLM2: When Smol Goes Big—Data-Centric Training of a Small Language Model.* 2025. arXiv:2502.02737.