LANGSELECT: TARGET-LANGUAGE ROUTING FOR LLM CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Code generators usually write in a programming language fixed before decoding. For function-level tasks that several languages can solve and the same unit tests can check, we study target-language routing under a verifier with LANGSELECT, a loop in which a policy orders candidate languages, a failed attempt falls back to the next language, and the code tokens of every attempt are counted. We build MultiLang-Bench, 3,000 tasks with verified solutions in eight languages. We show that the final pass rate depends only on which languages are tried and that, under a mild independence assumption, adjacent languages should be ordered by cost per success conditional on earlier failures. With GPT-5, a fixed language order emits 50.3% fewer code tokens than default-prompt Python on 450 held-out tasks (final pass rate 92.9% vs. 92.0%) and 48.2% fewer on a second, 400-task mix, where its pass rate is 2.8 points lower. A supervised selector and a LinUCB bandit that recover about 87% of the headroom in replay emit 3.7% and at least 13.9% more tokens than Python-only in live generation. A chain that ends in Python keeps the expected pass rate of Python-only; recomposed from logs, it saves 39% on the data used to choose it and 13% (95% CI -18 to 38%) in a fresh run on 60 of these tasks. Against a terse Python prompt, a cross-campaign estimate leaves the fixed order 11–15% and this chain no saving. With DeepSeek-Coder-6.7B, bandit routing emits 2.3 times as many tokens as Python-only. For GPT-5, generation time increases under routing, and reasoning tokens are not observed.

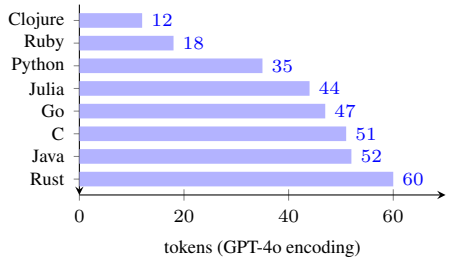
1 INTRODUCTION

LLM code generation increasingly runs inside verify-and-retry loops: a model writes code, tests execute it, and a failure triggers another attempt (Chen et al., 2023a; 2024; Jimenez et al., 2023; Yang et al., 2024). Every attempt adds generated tokens. Efficiency methods compress prompts (Jiang et al., 2023), shorten reasoning (Xu et al., 2025), or route queries between models (Chen et al., 2023b; Ding et al., 2024; Ong et al., 2024), but they keep the programming language of the output fixed. For many function-level tasks, such as standalone utilities, problem banks, and test oracles, several languages are acceptable and one test suite can check all of them. This paper asks when requesting a different language reduces the code a model emits once failed attempts and retries are counted.

Exploiting the variation shown in Figure 1 is harder than choosing the shortest language, for four reasons that our experiments quantify. First, a cheap language that fails costs its own tokens plus a retry. Second, replaying verified reference solutions fixes the pass rate at 100% and never pays for a retry, so replay can overstate what a router recovers. Third, the length of generated code depends on the requested language in ways that reference solutions do not show. Fourth, failures are correlated across languages: a task that fails in one language often fails in the next.

We study these effects with LANGSELECT, a verification-gated routing loop with a pluggable ranking policy, on MultiLang-Bench, a new corpus of 3,000 tasks with a verified solution in each of eight languages. We compare a fixed language order, a supervised selector, and LinUCB bandits in replay and in six live campaigns (Table 1): a main held-out comparison (MAIN), a second campaign on a different task mix (REP), a fresh run on a harder subset (HARD), partial-supervision tests (SUP), a prompt-style control (STYLE), and an open-weight generator (OPEN). Our main findings are the following.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.



Clojure: 12 tokens

```
(defn f [xs] (group-by first xs))
```

Python (the usual default): 35 tokens

```
from collections import defaultdict
def f(xs):
    d = defaultdict(list)
    for x in xs:
        d[x[0]].append(x)
    return dict(d)
```

Rust: 60 tokens

```
fn f(xs: Vec<&str>) -> HashMap<char, Vec<&str>>
    {
    let mut m = HashMap::new();
    for x in xs { m.entry(x.chars().next().
     unwrap()).or_insert(vec![]).push(x); }
    m }
```

Figure 1: A hand-written group-by function in eight languages ranges from 12 to 60 tokens. Across the 3,000 tasks of MultiLang-Bench, the longest verified solution of a task is on average 2.19× the shortest (95% CI $[2.17, 2.20]$). In live generation with GPT-5, however, most of the saving comes from the verbosity of the generated Python rather than from intrinsic concision (§5.2).

- **A fixed order reduces code tokens by about half on two task mixes with GPT-5.** A single fixed language order emits 50.3% fewer code tokens than default-prompt Python on 450 held-out tasks (final pass rate 92.9% vs. 92.0%). On a second mix of 400 tasks, where the logs show the order Ruby→Clojure→Julia, it emits 48.2% fewer (95% CI 40.9–55.0) at a pass rate 2.8 points lower (p=0.08). Savings are smaller on the MultiLang-Bench and McEval subsets of this mix (27–31%).

- **Replay overstates the savings of learned selectors.** In replay, the supervised selector and the LinUCB bandit recover 27.5 and 27.6 of the 31.8 points of hindsight headroom. In live generation, the supervised selector mostly requests Python and emits 3.7% more tokens than Python-only, although it has the highest final pass rate (93.8%). The warm-started bandit emits at least 13.9% more tokens than Python-only and passes fewer tasks (85.0% vs. 91.0%).

- **The fallback chain, not only the first choice, determines pass rate and cost.** The final pass rate depends only on which languages are tried, and adjacent languages should be ordered by cost per success conditional on earlier failures (Proposition 1). After Ruby and Clojure failed, Python passed 22 of 47 tasks. Recomposed from logged attempts, the chain Ruby→Clojure→Python solves 11 more tasks than Python-only and emits 39.1% fewer tokens on the campaign used to choose it; in a fresh run on 60 of these tasks, it saves 13.2% (95% CI −17.6 to 37.9).

- **The savings depend on prompt style, task difficulty, and generator.** A terse Python prompt alone removes 41.6% of default Python tokens. Against it, a cross-campaign estimate leaves the fixed order 11–15% and the Python-ending chain no saving. For DeepSeek-Coder-6.7B, bandit routing more than doubles code tokens because its non-Python attempts rarely pass. For GPT-5, generation time increases under routing, and reasoning tokens are not observed.

Contributions. (1) A formulation of target-language routing under a verifier, with a decomposition of counted tokens into first-attempt, wasted, and fallback parts, and an exchange argument that separates which languages to try (final pass rate) from their order (cost). (2) MultiLang-Bench, a 3,000-task corpus with verified solutions in eight languages and fixed splits. (3) A replay-versus-live comparison of fixed, supervised, and bandit routers with paired statistics and component ablations, together with experiments on prompt style, harder tasks, partial supervision, and an open-weight generator. Throughout, we measure emitted code tokens; §5.5 discusses what this measure omits. Code is available at `https://anonymous.4open.science/r/LangSelect-130E`.

2 TARGET-LANGUAGE ROUTING UNDER A VERIFIER

Setting. A task t has a language-neutral specification s_t and a test suite $\mathcal{U}_t$ that can check an implementation in any language of a portfolio $\mathcal{L} = \{\ell_1, \ldots, \ell_K\}$. A fixed generator $\mathcal{G}$ is prompted to write the task in a requested language ℓ. The verifier accepts an attempt if it passes $\mathcal{U}_t$ and a language-fidelity check (it parses or compiles as ℓ and defines the expected entry point), which rejects outputs that drift into another language (Moumoula et al., 2025). For attempt ℓ on task t, let $c_{t\ell} \geq 0$ be the number of emitted tokens and $z_{t\ell} \in \{0, 1\}$ the verifier outcome.

Policies and chains. A routing policy π maps s_t, using only information available before generation, to an ordered chain $\sigma_t = (\sigma_1, \ldots, \sigma_m)$ of distinct languages with $m \leq B$ (we use a budget of B=3 attempts). The loop tries σ_1, stops at the first accepted attempt, and otherwise falls back to the next language. Every attempt is counted:

$$C_t(\sigma) = \sum_{k=1}^{m} c_{t\sigma_k} \prod_{j<k} \big(1 - z_{t\sigma_j}\big), \qquad Z_t(\sigma) = 1 - \prod_{k=1}^{m} \big(1 - z_{t\sigma_k}\big), \tag{1}$$

where Z_t indicates whether any attempt passes; we call its mean the *final pass rate*. Splitting C_t by the role of each attempt gives the identity

$$C_t = \underbrace{c_{t\sigma_1} z_{t\sigma_1}}_{\text{first-attempt success } F_t} + \underbrace{\textstyle\sum_{k:\, z_{t\sigma_k}=0} c_{t\sigma_k} \prod_{j<k} (1 - z_{t\sigma_j})}_{\text{wasted } W_t} + \underbrace{\textstyle\sum_{k\geq 2} c_{t\sigma_k} z_{t\sigma_k} \prod_{j<k} (1 - z_{t\sigma_j})}_{\text{successful fallback } S_t}, \tag{2}$$

and we call $\mathbb{E}[W_t]/\mathbb{E}[C_t]$ the *waste share*. With d the default language (Python) and $q_d = \mathbb{E}[z_{td}]$ its pass rate, the routing problem is

$$\min_{\pi} \ \mathbb{E}_t\big[C_t(\pi(s_t))\big] \quad \text{s.t.} \quad \mathbb{E}_t\big[Z_t(\pi(s_t))\big] \ \geq \ q_d - \epsilon. \tag{3}$$

Replay versus live. *Verified-solution replay* evaluates a policy by counting, for its first language, the length of an already verified reference solution, so $z \equiv 1$, $\epsilon = 0$ by construction, and no fallback is ever needed. *Live generation* samples fresh outputs and counts every attempt as in Eq. (1). Replay measures the headroom in a verified corpus; only live generation measures what a policy recovers.

What makes a good chain. Eq. (1) separates two decisions. Let $F_{<k}$ denote the event that the first $k-1$ attempts fail, and write $\bar{c}_\ell^{(k)} = \mathbb{E}[c_{t\ell} \mid F_{<k}]$ and $\rho_\ell^{(k)} = \Pr[z_{t\ell}=1 \mid F_{<k}]$ for the conditional cost and conditional success of language ℓ at position k.

Proposition 1 (The set determines pass; conditional cost per success determines adjacent order)**.** *(a) $Z_t(\sigma)$ depends only on the set of languages in σ, not on their order. In particular, if the default d is in the chain, then $Z_t(\sigma) \geq z_{td}$ for every realization, where z_{td} is the outcome of the chain's own attempt in d, and $\mathbb{E}[Z_t(\sigma)] \geq q_d$ whenever that attempt is sampled as in the default policy. (b) Suppose that, given $F_{<k}$, the cost of one language's attempt is independent of whether another language's attempt passes. Then placing i immediately before j at positions $k, k{+}1$ gives expected cost no larger than the reverse order iff $\bar{c}_i^{(k)}\, \rho_j^{(k)} \leq \bar{c}_j^{(k)}\, \rho_i^{(k)}$, i.e., iff i has the smaller conditional cost per success $\bar{c}^{(k)}/\rho^{(k)}$. (c) If, in addition, outcomes are independent across languages, then for any fixed set the expected cost is minimized by sorting languages by $\bar{c}_\ell/p_\ell$, where $\bar{c}_\ell = \mathbb{E}[c_{t\ell}]$ and $p_\ell = \Pr[z_{t\ell}=1]$, the classical index rule for sequential search (Simon & Kadane, 1975).*

The proof is a short exchange argument (Appendix A). Three consequences guide the rest of the paper. First, if the default language ends the chain, the pass constraint in Eq. (3) holds with $\epsilon = 0$ in expectation, and only cost remains to be decided: the chain $(\ell, \ldots, d)$ costs less than d alone iff $\bar{c}_\ell + \Pr[z_{t\ell}=0]\, \mathbb{E}[C_{\text{rest}} \mid z_{t\ell}=0] < \bar{c}_d$, where C_{rest} is the cost of the remainder of the chain. Second, the next language to try should have the lowest cost per success *given that the earlier languages failed*; when failures are correlated, this ranking can differ from the marginal one, and a language that rarely succeeds after the earlier failures should be dropped. Third, these quantities are conditional on the chain prefix, whereas a per-attempt bandit estimates a marginal reward for each language; this is one reason why learned first-choice selectors need not reduce the counted tokens (§5.3).

3 LANGSELECT: A VERIFICATION-GATED ROUTING LOOP

LANGSELECT (Figure 2) implements Eq. (1). All policies share the prompt template, generator settings, verifier, token counter, and budget of B=3 attempts; only the chain differs. Python

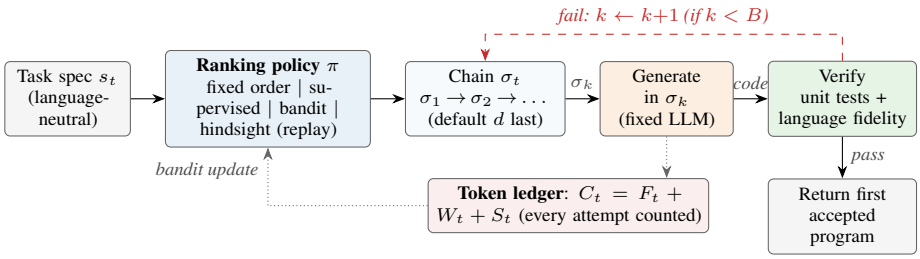


Figure 2: LANGSELECT. A policy turns the task into an ordered chain of languages; each attempt is verified, a failure falls back to the next language, and every attempt is counted. Only bandit policies update from the ledger.

generations are cached per task, so every policy that reaches Python on a task reuses the same Python draw (Appendix C). Any chain that differs from a logged chain only in where Python appears can therefore be evaluated exactly from logged attempts. We study the following policies.

Fixed order. One global order applied to every task, fixed before the live campaigns from training-side statistics (Appendix D). In every campaign with per-attempt logs, it requests Ruby, then Clojure, then Julia. Proposition 1 motivates the variant Ruby→Clojure→Python, which we evaluate by recomposing logged attempts (§4).

Domain order (replay only). The same rule computed separately for each task domain (e.g., Clojure for math, Ruby for string, and Python for numeric tasks).

Supervised selector. A classifier over a frozen CodeBERT embedding (Feng et al., 2020) of s_t concatenated with metadata available before generation (source group, domain, difficulty, and argument/return type family), trained on the training split to predict the cheapest verified language. Its class scores order the chain. Six further supervised classifiers on the same features are compared in Appendix E.

LinUCB bandit. Learned policies are trained on the training split, and the bandit is additionally warm-started on a 377-task training-side corpus with per-language costs (Appendix D). The bandit uses per-language linear UCB heads (Li et al., 2010) on a 256-dimensional task representation produced by a supervised MLP over CodeBERT features (Zhou et al., 2020). The heads start from the fixed-order prior, which the UCB score overrides when it is higher. After each attempt, only the head of the attempted language is updated, with reward $r = (1 - \tau(c))\,\mathbf{1}[\text{pass}]$, where τ maps token counts to $[0, 1]$ by corpus percentiles. As implemented, this policy also generates a Python candidate in parallel on every task and counts its tokens. We therefore report both this *as-run* cost and a *sequential* cost that counts only the attempts actually used (Appendix D).

Hindsight. The cheapest verified language of each test task. In replay it is a reference within the corpus, not an upper bound on live behavior; in live campaigns it only supplies the first language, and we do not treat it as an oracle (§5.2).

Deployment constraints. A hard mask restricts chains to an allowed set $A_t \subseteq \mathcal{L}$, and penalties for maintainability or integration cost can be subtracted from any policy's scores (Appendix E). Selection adds no time for the fixed order and about 0.24 s per task for the learned policies in our logs, mostly for the CodeBERT embedding, compared with about 15 s for one GPT-5 generation.

4 EXPERIMENTAL SETUP

MultiLang-Bench. We build a corpus of 3,000 deterministic pure-function tasks. Each task has a language-neutral prompt, deterministic tests, and one accepted solution in each of C, Clojure, Go, Java, Julia, Python, Ruby, and Rust; all 24,000 solutions pass the test harness and a language-fidelity check. The tasks come from HumanEval/MBPP/APPS-style sources (990), Codeforces/LeetCode-style sources (1,230), and manually written or author-reviewed tasks (780), with a fixed 2,100/450/450 train/validation/test split. Exact, near-duplicate, and semantic overlaps with HumanEval-X, MultiPL-E, and McEval are removed (Appendix B). Requiring a solution in all eight languages biases the

Table 1: Evaluation campaigns. All live campaigns use GPT-5 except OPEN. MLB: MultiLang-Bench; HE-X: HumanEval-X; MPL-E: MultiPL-E. Policies are compared within a campaign; §5.5 combines ratios across campaigns only as a labeled estimate.

Name	Tasks	n	Policies	Purpose
Replay	MLB test split	450	Python-only, domain order, supervised, CodeBERT-only, LinUCB bandit, hindsight	headroom at fixed correctness
MAIN	MLB test split	450	Python-only, fixed order, supervised, hindsight	main live comparison
REP	HE-X 50, MPL-E 114, McEval 50, MLB 186	400	Python-only, fixed order, LinUCB bandit and two ablations, hindsight	second task mix; bandit ablation; token decomposition; recomposed chains
HARD	60 of the REP tasks (McEval 30, MLB 30)	60	fixed order, supervised, LinUCB bandit with and without fallback	fresh run on a harder subset; case studies
SUP	(a) the REP tasks; (b) 236 McEval and MLB tasks	400; 236	LinUCB bandit warm-started with (a) $r \in \{2, 4, 6, 8\}$ revealed languages per task or (b) 25–100% of warm-start tasks	supervision requirements
STYLE	MLB held-out tasks, 3 buckets of 150	450	Python with default prompt, terse prompt, terse prompt and AST compaction	prompt-style control
OPEN	HE-X, MPL-E, McEval, MLB (20 each)	80	DeepSeek-Coder-6.7B: bandit choice with Python fallback	open-weight generator

corpus toward functions that every language can express; this requirement enables per-task replay and limits the scope of our claims.

Campaigns. Table 1 lists the replay study and the six live campaigns. MAIN is the main held-out comparison. REP samples GPT-5 again on a different 400-task mix and ablates the bandit. HARD reruns 60 of the REP tasks from McEval and MultiLang-Bench with fresh generations for every language, including Python; these tasks are harder than average (Python-only passes 81.7% of them in REP, against 91.0% overall), and the run adds a no-fallback bandit. The campaigns differ in task mix, API route, and date, so absolute pass rates differ across them (Python-only passes 92.0% of MAIN and 91.0% of REP). MAIN is reported from aggregate statistics; the other campaigns have per-attempt logs, which we use for paired tests and recomposition.

Token counts. We count *code tokens*: all tokens of the emitted code string, including any signature, imports, and wrapper that the model writes, summed over every attempt (the harness-proxy count in our logs). Function-body counts are also logged, but the body extractor is unreliable for several languages (Appendix C), so we do not report live body-level savings. Live counts use the public `cl100k_base` encoding for every language and policy, and corpus statistics use the GPT-4o encoding. Neither is the generator's billing tokenizer, and GPT-5's hidden reasoning tokens are not in our logs (§5.5).

Statistics and recomposition. We report Wilson 95% intervals for pass rates, paired bootstrap intervals (10,000 task resamples) for the relative saving $1-\sum_t C_t(\pi)/\sum_t C_t(d)$, and exact McNemar tests for paired pass differences, where b (c) counts tasks solved only by the policy (only by Python-only). Chains that replace or append Python in the logged fixed-order chain are evaluated by *recomposition*: for each task, we reuse the logged Ruby and Clojure attempts and, where the new chain reaches Python, the task's cached Python draw. Recomposed results are exact for these logged draws but are not fresh generations, and we label them as such.

5 RESULTS

5.1 REPLAY HEADROOM

In replay over the 450 test tasks (function-body tokens of reference solutions; pass fixed at 100%), choosing each task's cheapest language (hindsight) uses 31.8% fewer tokens than Python-only. The LinUCB bandit recovers 27.6 points of this headroom and the supervised selector 27.5, compared with 20.0 for the domain order and 18.9 for a CodeBERT-only classifier (Appendix E). Headroom varies with the workload: it is 41.2% for math and 41.5% for string tasks, but 8.8% for numeric tasks. On external benchmarks, the bandit's replay saving is 28.3% (HumanEval-X), 31.6% (MultiPL-E), and 34.2% (McEval). Measured in code tokens on the 377 warm-start tasks, hindsight headroom is

Table 2: Live comparison with GPT-5. Tokens: code tokens per task, all attempts counted. First and Final: pass rate at the first attempt and after fallback (Wilson 95% CI). Savings are relative to Python-only in the same campaign. For REP, the saving interval is a paired bootstrap interval, and b/c counts tasks solved only by the policy / only by Python-only (exact McNemar p). MAIN is reported from aggregate statistics.

	MAIN: MultiLang-Bench test split (n=450)				REP: second task mix (n=400)			
Policy	First	Final	Tokens	Saving	Final	Tokens	Saving [95% CI]	b/c (p)
Python-only	92.0	92.0 [89.1, 94.2]	141.4	—	91.0 [87.8, 93.4]	161.7	—	—
Fixed order	87.3	92.9 [90.1, 94.9]	**70.3**	**50.3%**	88.3 [84.7, 91.0]	**83.7**	**48.2%** [40.9, 55.0]	11/22 (.08)
Supervised selector	91.8	**93.8** [91.2, 95.7]	146.7	−3.7%	not run			

Table 3: Learned selectors and their ablations (code tokens per task). *As run*: including the bandit's parallel Python candidate. *Sequential*: only the attempts actually used. Savings use sequential counts and are relative to Python-only in the same campaign (in HARD, the cached Python draw), with paired bootstrap 95% CIs.

	Policy	First	Final	As run	Sequential	Saving [95% CI]	b/c (p)
REP n=400	Python-only	91.0	91.0	161.7	161.7	—	—
	Fixed order	80.5	88.3	83.7	83.7	**48.2%** [40.9, 55.0]	11/22 (.08)
	LinUCB bandit	76.3	85.0	320.3	184.1	−13.9% [−31.6, 2.0]	9/33 (<.001)
	without warm start	73.5	90.3	241.1	125.7	22.3% [12.9, 31.4]	9/12 (.66)
	without task features	45.0	77.3	511.4	349.7	−116.3% [−141.1, −94.0]	8/63 (<.001)
HARD n=60	Python-only	83.3	83.3	225.0	225.0	—	—
	Fixed order	55.0	75.0	153.7	153.7	31.7% [7.9, 49.9]	1/6 (.13)
	Supervised selector	76.7	83.3	278.5	278.5	−23.8% [−45.7, −9.0]	0/0 (1.0)
	LinUCB bandit, no fallback	41.7	41.7	143.3	143.3	36.3% [15.6, 50.8]	0/25 (<.001)
	LinUCB bandit	48.3	75.0	478.8	335.3	−49.1% [−91.5, −14.2]	0/5 (.06)

19.1%, or 15.2% when the portfolio is restricted to Python, Java, Go, and Ruby. Replay therefore ranks the learned selectors well above a coarse domain rule (87% vs. 63% of the headroom). In live generation, however, neither learned selector emits fewer tokens than a single fixed order (§5.2–5.3).

5.2 LIVE COMPARISON WITH PYTHON-ONLY

Table 2 reports the live results. In MAIN, the fixed order emits 50.3% fewer code tokens than Python-only, with a final pass rate of 92.9% versus 92.0%; it falls back on 12.4% of tasks and uses 1.20 attempts per task. In REP, the saving is similar (48.2%, 95% CI 40.9–55.0), but the final pass rate is 2.8 points lower than that of Python-only (22 tasks solved only by Python-only, 11 only by the fixed order; p=0.08). The saving is not driven by a few long outputs: per task, the fixed order emits fewer code tokens on 347 of the 400 tasks (median per-task saving 57.1%, 10th percentile −21.2%). The Python baseline is also stable: SUP drew Python again on the same 400 tasks, and the two draws agree on pass or fail for 394 tasks, with 161.7 and 163.7 mean code tokens (per-task Pearson r=0.81). The agreement of the two aggregate savings partly reflects task mix: on the 186 MultiLang-Bench tasks of REP, the saving is 27.0% (Table 5). In REP and HARD, the fixed order requests Ruby first on every task. The supervised selector reaches the highest final pass rate in MAIN (93.8%) but emits 3.7% more tokens than Python-only; in HARD it requests Python first on 53 of 60 tasks. We omit the live hindsight row: in REP it had no per-task labels and reduces to the fixed order, and in MAIN its attempt profile matches that of the fixed order (Appendix F).

Source of the saving. The saving reflects how the generator writes each language more than how concise the languages are. In the verified corpus, Python and Ruby reference solutions have similar mean lengths (43.9 and 45.3 body tokens). On the 317 REP tasks where both first attempts pass, however, GPT-5 emits 133.8 code tokens in Python and 44.0 in Ruby, and the Ruby output is shorter on 95.9% of these tasks. This difference explains why the live saving (48–50%) exceeds the replay headroom measured in code tokens (19.1%): replay compares reference solutions, whereas live generation compares the model's own outputs. It also implies that a router trained on the lengths of reference solutions learns the corpus's verbosity rather than the generator's.

Table 4: Decomposition of code tokens per task (Eq. (2)) for alternative chains. R/C/J/Py: Ruby/Clojure/Julia/Python. Waste: W as a share of the total. †Recomposed from logged attempts (§4); other rows as run.

	Chain	Final	First F	Wasted W	Fallback S	Total	Waste	Saving [95% CI]	b/c (p)
REP n=400	Python-only	91.0	138.8	22.9	0.0	161.7	14%	—	—
	R†	80.5	35.7	15.5	0.0	51.2	30%	68.3% [65.1, 71.4]	5/47 (<.001)
	R→C†	88.3	35.7	29.9	7.8	73.5	41%	54.6% [48.8, 59.9]	11/22 (.08)
	R→C→J (as run)	88.3	35.7	40.2	7.8	83.7	48%	48.2% [40.9, 55.0]	11/22 (.08)
	R→Py†	92.3	35.7	36.4	32.7	104.8	35%	35.2% [28.3, 42.4]	5/0 (.06)
	R→C→Py†	93.8	35.7	45.2	17.6	98.5	46%	39.1% [29.7, 48.0]	11/0 (.001)
HARD n=60	Python-only	83.3	203.1	21.9	0.0	225.0	10%	—	—
	R→C→J (as run)	75.0	30.9	94.0	28.7	153.7	61%	31.7% [7.9, 49.9]	1/6 (.13)
	R→C→Py†	85.0	30.9	92.2	72.1	195.3	47%	13.2% [−17.6, 37.9]	1/0 (1.0)

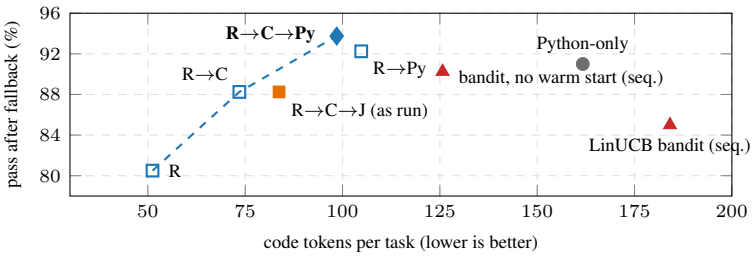


Figure 3: Code tokens and final pass rate in REP (n=400). Open squares and the diamond are chains recomposed from logged attempts (in-sample); the dashed line connects the best observed points. The two chains that end in Python reach or exceed the pass rate of Python-only with 35–39% fewer tokens. Bandit points use sequential counts.

5.3 LEARNED SELECTORS

Table 3 isolates the learned components. The warm-started LinUCB bandit passes fewer tasks than Python-only (85.0% vs. 91.0%, p<0.001) and emits 13.9% more tokens even under sequential counting. Its first choices are mostly Go (252 of 400 tasks) and Python (122). Go passes 71.8% of first attempts at 108 tokens, or 151 tokens per success, compared with 64 for Ruby. Without the warm start, the bandit shifts toward Julia and Python and saves 22.3% at a final pass rate not significantly different from that of Python-only (p=0.66), which is still less than the fixed order saves. Without task features, it requests C for every task. In HARD, a no-fallback variant of the bandit passes 41.7% of tasks; the full policy passes 48.3% at the first attempt and 75.0% after fallback, at 192 more sequential tokens per task than the no-fallback variant. The supervised selector passes more tasks than the bandit with fewer tokens, but it requests Python first on most tasks.

Replay favored the learned selectors because it counts only the extra length of a wrong first choice, whereas live generation also counts its failures and fallbacks; in addition, the generator's verbosity differs from the corpus labels on which the selectors were trained (§5.2). Supervision is not the limiting factor. When warm-started on labels generated by GPT-5 itself (SUP), the bandit requests Ruby first on 61–72% of tasks and approaches the fixed order. With only two of eight languages revealed per task, it reaches a final pass rate of 89.5% at 114 sequential tokens per task, compared with 87.8% and 101 tokens when all eight are revealed (Appendix H). The fixed order itself requires only per-language cost and pass statistics, which partial attempt logs provide.

5.4 FALLBACK CHAINS

Table 4 decomposes the counted tokens. The as-run fixed order spends 48% of its tokens on failed attempts, yet it emits 83.7 tokens per task against 161.7 for Python-only, because first-attempt Ruby successes (80.5% of tasks) are short. Failures are strongly correlated across languages. Python passes 98.4% of the tasks on which Ruby passes, but only 60.3% of those on which Ruby fails. After Ruby fails (78 tasks), Clojure passes 39.7%. After Ruby and Clojure both fail (47 tasks), Julia passes none, whereas the cached Python draw passes 22. Julia's result may partly reflect an entry-point defect in the harness (Appendix C); the Python result does not depend on it. The HARD case studies show the

Table 5: Results by benchmark in REP (code tokens). Ruby first: first-attempt pass rate of Ruby. Savings are relative to Python-only on the same tasks; R→C→Py is recomposed.

		Python-only			Fixed order (as run)		R→C→Py	
Benchmark	n	Final	Tokens	Ruby first	Final	Saving	Final	Saving
HumanEval-X	50	94.0	173.4	96.0	98.0	74.2%	98.0	61.4%
MultiPL-E	114	98.2	182.8	93.9	96.5	70.9%	99.1	69.2%
MultiLang-Bench	186	86.0	113.9	80.6	84.4	27.0%	89.8	20.5%
McEval	50	90.0	279.6	34.0	74.0	30.6%	92.0	8.5%

same pattern: on each of the six tasks that the supervised selector solves and the fixed order does not, Ruby, Clojure, and Julia all fail and Python passes (Appendix F).

Proposition 1 turns these observations into a chain. Among the languages logged at each position, the estimated conditional cost per success is 64 tokens for Ruby and 178 for Python at position 1, 287 for Clojure and 456 for Python at position 2, and 456 for Python at position 3, where Julia never succeeds (Appendix F, Table 15). The adjacent-swap condition therefore selects Ruby→Clojure→Python. Recomposed from the logged attempts, this chain solves 375 of 400 tasks, compared with 364 for Python-only, and emits 39.1% fewer tokens (Figure 3). It loses no task relative to Python-only by construction, because it reuses the Python-only draw (Proposition 1(a)); the McNemar test therefore adds no evidence here. Removing Julia without adding Python (Ruby→Clojure) keeps the as-run pass rate of 88.3% and raises the saving to 54.6%. Because the conditional rates and the evaluation come from the same campaign, these estimates are optimistic. HARD provides fresh generations on 60 of these tasks. There, Ruby→Clojure→Python passes 85.0% of tasks against 83.3% for Python-only but saves only 13.2% (95% CI −17.6 to 37.9), whereas the as-run fixed order loses 8.3 points of pass rate. On the same 60 tasks, the recomposition from REP gives a 21.4% saving, so the lower figure mainly reflects the harder tasks, with additional sampling variation.

5.5 SENSITIVITY TO PROMPT STYLE, WORKLOAD, AND GENERATOR

Prompt style. Part of the saving is an effect of style. In STYLE, a terse prompt that adds instructions to prefer comprehensions and built-ins and to use short names reduces GPT-5's default Python from 145.9 to 85.1 code tokens per task (41.6% fewer), with a single-attempt pass rate of 65% instead of 68%; AST compaction reaches 43.4% fewer tokens with a 55% pass rate. The absolute pass rates of STYLE are not comparable with those of MAIN (Appendix G). Composing ratios across campaigns, the fixed order would save about 11–15% relative to terse Python, and Ruby→Clojure→Python would use about 4% more tokens than terse Python. Output language and prompt style therefore overlap as sources of savings: compressing style lowers the pass rate, whereas routing with Python last does not. We have not tested routing combined with terse prompts in every language.

Workload. Table 5 shows that the saving follows the first-attempt success of Ruby. On HumanEval-X and MultiPL-E, where Ruby passes 94–96% of first attempts, Ruby→Clojure→Python saves 61–69% at an equal or higher pass rate. On McEval, where Ruby passes 34% of first attempts, the as-run fixed order loses 16 points of pass rate, and the chain that ends in Python recovers the pass rate but saves only 8.5%. This pattern is consistent with the break-even condition of Section 2.

Generator. With DeepSeek-Coder-6.7B-Instruct (OPEN, 80 tasks), the bandit's non-Python choices pass on 9 of 80 tasks (Rust 6/45, Ruby 3/17, Go 0/11, Java 0/4), and inspected failures contain Python syntax, the programming-language confusion described by Moumoula et al. (2025). With Python as fallback, the pass rate is 61.3% versus 57.5% for Python-only (3 tasks gained, none lost), but even if Python is generated only after a failure, the chain emits 592.0 code tokens per task against 256.8, or 2.3 times as many. Routing helps only when the generator writes the requested language reliably.

Unmeasured costs. We count emitted code tokens. GPT-5 also bills hidden reasoning tokens, which our logs do not record, and each additional attempt resends the prompt. Wall-clock time is a rough proxy for all generated tokens. In REP, the fixed-order chain took 27.9 s per task on average, against 14.9 s for the Python draw (medians 15.9 and 9.5 s), and the Ruby first attempt took longer than the Python draw on 259 of 400 tasks. The Python draws were generated at a different time, so API load may confound this comparison; the direction is consistent with longer hidden reasoning for less common languages. The reported savings therefore concern emitted code, not billed cost.

6 Related Work

Cost-aware LLM inference. Prompt compression (Jiang et al., 2023) and concise reasoning (Xu et al., 2025) reduce what the model reads or generates while reasoning, and cascades and routers send a query to a cheaper model when it suffices (Dohan et al., 2022; Chen et al., 2023b; Ding et al., 2024; Ong et al., 2024). These methods keep the output format fixed. We keep the model fixed and vary the output language, so the two approaches could be combined; we have not measured such a combination (Appendix F).

Verification in the loop. Execution feedback is central to current code generation, from test-based reranking (Chen et al., 2023a) and self-debugging (Chen et al., 2024) to repository-level agents (Jimenez et al., 2023; Yang et al., 2024). The token decomposition in Eq. (2) applies to any such loop. Proposition 1 is an instance of the classical least-cost sequential search problem (Simon & Kadane, 1975); our data indicate that, when failures are correlated, its index rule should be applied to conditional rather than marginal success rates.

Multilingual code generation. HumanEval-X, MultiPL-E, and McEval measure whether a model solves tasks across programming languages (Zheng et al., 2023; Cassano et al., 2023; Chai et al., 2024), building on HumanEval, MBPP, and APPS (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021), and code models generate in many languages (Nijkamp et al., 2022; Li et al., 2023; Rozière et al., 2023). We ask instead which acceptable language to request first and what to request after a failure. Models often drift toward Python when asked for another language (Moumoula et al., 2025), a programming-language counterpart of language confusion in natural-language generation (Marchisio et al., 2024); for the 6.7B model we tested, it more than doubled counted tokens.

Language-dependent token counts. Tokenizers segment natural languages unevenly, so the same content requires more tokens in some languages (Petrov et al., 2023; Ahia et al., 2023), and programming languages differ in concision (Nanz & Furia, 2015). For GPT-5, we find a larger effect from language-dependent style: Python and Ruby reference solutions in our corpus are about equally long, whereas GPT-5's own Python output is about three times as long as its Ruby output.

7 Discussion and Conclusion

Scope. Our results concern function-level, unit-test-verifiable tasks that several languages can solve. They do not apply when the repository or deployment fixes the language, when integration, dependencies, or maintainability dominate the choice of language, or when outputs are repository-level edits. A portfolio restricted to Python, Java, Go, and Ruby keeps 15.2% of the 19.1% replay headroom (Appendix E). The live results come from GPT-5 and one 6.7B open-weight model, and tokens are counted with a public encoding rather than the billing tokenizer.

Limitations. The Ruby→Clojure→Python chain is recomposed from logged attempts and needs a prospective run, the style control comes from a separate campaign, and GPT-5's reasoning tokens are not observed. Appendix J lists further limitations, and Appendix I links every reported number to its campaign.

Conclusion. With GPT-5, the requested output language substantially changes the length of the code the model emits: a fixed language order emits about half as many code tokens as default-prompt Python on two task mixes, with smaller savings on harder subsets. Replay over reference solutions overstates what learned selectors recover in live generation, where failed attempts and fallbacks are counted and the generator's style differs from that of the reference solutions. The fallback chain, not only the first choice, determines pass rate and cost: the final pass rate depends only on which languages are tried, and ending the chain in Python keeps the expected pass rate of Python-only. Routing did not save tokens for one 6.7B open-weight model, and the savings remain untested against billed reasoning tokens and a same-campaign terse-prompt baseline. We therefore recommend evaluating routing live, counting every attempt, against a style-controlled baseline.

AI Use Statement

In this work, we used generative AI tools for language editing, for restructuring the manuscript and drafting revised text from the authors' results and notes, and for writing scripts that re-analyze logged

experiment results (Wilson and bootstrap confidence intervals, exact McNemar tests, token-count decomposition, and recomposition of logged attempts). Reported numbers were cross-checked against the raw experiment logs. GPT-5 and DeepSeek-Coder-6.7B-Instruct are the objects of study and generated the code that we evaluate. We take responsibility for the final content of this work.

Ethics Statement

This work involves no human subjects, annotators, or personal data. MultiLang-Bench is built from public programming benchmarks and contest-style problems rewritten as language-neutral prompts, together with manually written tasks; each task records its source and license field, and prompts are rewritten rather than redistributed verbatim. Generated code is executed in temporary directories under fixed timeouts; the released harness does not provide OS-level sandboxing, so we recommend running it inside a container. Target-language routing can reduce the number of code tokens a model emits; we caution that token savings are not a billing forecast (§5.5) and that deployments must weigh maintainability and team expertise, which our experiments do not measure.

Reproducibility Statement

Section 4 and Appendices B–C specify the corpus, splits, harnesses, prompts, attempt budget, and accounting boundaries; Appendix D gives selector architectures and hyperparameters; Appendix I maps every reported number to its campaign and log file. The anonymous repository (`https://anonymous.4open.science/r/LangSelect-130E`) contains the routing loop, the selectors, the language harnesses, the live evaluator and ablation scripts (`scripts/run_*.py`), the warm-start corpus, the trained context-head weights, and step-by-step instructions in `REPRODUCE.md`.

References

Orevaoghene Ahia, Sachin Kumar, Hila Gonen, Jungo Kasai, David Mortensen, Noah Smith, and Yulia Tsvetkov. Do all languages cost the same? tokenization in the era of commercial language models. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 9904–9923, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.614. URL https://aclanthology.org/2023.emnlp-main.614/.

J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton. Program synthesis with large language models, 2021.

F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, A. Guha, M. Greenberg, and A. Jangda. MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 2023. doi: 10.1109/TSE.2023.3267446.

L. Chai, S. Liu, J. Yang, Y. Yin, K. Jin, J. Liu, T. Sun, G. Zhang, C. Ren, H. Guo, Z. Wang, B. Wang, X. Wu, B. Wang, T. Li, L. Yang, S. Duan, and Z. Li. McEval: Massively multilingual code evaluation, 2024.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *International Conference on Learning Representations (ICLR)*, 2023a.

L. Chen, M. Zaharia, and J. Zou. FrugalGPT: How to use large language models while reducing cost and improving performance, 2023b.

M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr,

J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code, 2021.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations (ICLR)*, 2024.

Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Rühle, Laks V. S. Lakshmanan, and Ahmed Hassan Awadallah. Hybrid LLM: Cost-efficient and quality-aware query routing. In *International Conference on Learning Representations (ICLR)*, 2024.

D. Dohan, W. Xu, A. Lewkowycz, J. Austin, D. Bieber, R. G. Lopes, Y. Wu, H. Michalewski, R. A. Saurous, J. Sohl-Dickstein, K. Murphy, and C. Sutton. Language model cascades, 2022.

Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Proc. EMNLP Findings*, 2020. doi: 10.18653/v1/2020.findings-emnlp.139.

D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt. Measuring coding challenge competence with APPS. In *Proc. NeurIPS Datasets and Benchmarks*, 2021.

H. Jiang, Q. Wu, C.-Y. Lin, Y. Yang, and L. Qiu. LLMLingua: Compressing prompts for accelerated inference of large language models. In *Proc. EMNLP*, 2023. doi: 10.18653/v1/2023.emnlp-main.825.

C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues?, 2023.

L. Li, W. Chu, J. Langford, and R. E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proc. WWW*, 2010. doi: 10.1145/1772690.1772758.

R. Li, L. Ben Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. Muñoz Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries. StarCoder: May the source be with you!, 2023.

Kelly Marchisio, Wei-Yin Ko, Alexandre Berard, Théo Dehaze, and Sebastian Ruder. Understanding and mitigating language confusion in LLMs. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 6653–6677, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.380. URL https://aclanthology.org/2024.emnlp-main.380/.

M. B. Moumoula, S. L. Nikiema, A. K. Kabore, J. Klein, and T. F. Bissyande. Programming language confusion: When code LLMs can't keep their languages straight, 2025. Accepted at SANER 2026.

S. Nanz and C. A. Furia. A comparative study of programming languages in Rosetta Code. In *Proc. ICSE*, 2015. doi: 10.1109/ICSE.2015.90.

E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong. CodeGen: An open large language model for code with multi-turn program synthesis, 2022.

I. Ong, A. Almahairi, V. Wu, W.-L. Chiang, T. Wu, J. E. Gonzalez, M. W. Kadous, and I. Stoica. RouteLLM: Learning to route LLMs with preference data, 2024.

A. Petrov, E. La Malfa, P. H. S. Torr, and A. Bibi. Language model tokenizers introduce unfairness between languages, 2023.

B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. Canton Ferrer, A. Grattafiori, W. Xiong, A. Dèfossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve. Code Llama: Open foundation models for code, 2023.

Herbert A. Simon and Joseph B. Kadane. Optimal problem-solving search: All-or-none solutions. *Artificial Intelligence*, 6(3):235–247, 1975.

S. Xu, W. Xie, L. Zhao, and P. He. Chain of draft: Thinking faster by writing less, 2025.

J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. SWE-agent: Agent-computer interfaces enable automated software engineering, 2024.

Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li, T. Su, Z. Yang, and J. Tang. CodeGeeX: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X. In *Proc. KDD*, 2023. doi: 10.1145/3580305.3599790.

D. Zhou, L. Li, and Q. Gu. Neural contextual bandits with UCB-based exploration. In *Proc. ICML*, 2020.

A PROOFS FOR SECTION 2

We use the notation of Section 2. Fix a task and a chain $\sigma = (\sigma_1, \ldots, \sigma_m)$; write $z_k = z_{t\sigma_k}$, $c_k = c_{t\sigma_k}$, and $\mathbf{1}_{F_{<k}} = \prod_{j<k}(1 - z_j)$, so that $C_t(\sigma) = \sum_k c_k \mathbf{1}_{F_{<k}}$.

Decomposition (Eq. (2)). Each term $c_k \mathbf{1}_{F_{<k}}$ is nonzero only if attempt k is made. If $z_k = 1$, the attempt is the returned one: it contributes to F_t when $k = 1$ and to S_t when $k \geq 2$. If $z_k = 0$, it contributes to W_t. The three sums partition the terms, so $C_t = F_t + W_t + S_t$ holds exactly on every task; our logs satisfy this identity to rounding (Table 4).

Proof of Proposition 1(a). $1 - Z_t(\sigma) = \prod_k (1 - z_k)$ is a product over the set $\{\sigma_1, \ldots, \sigma_m\}$ and is invariant to order. If d is in the set, then $\prod_k (1 - z_k) \leq 1 - z_{td}$, so $Z_t(\sigma) \geq z_{td}$ on every realization; taking expectations gives $\mathbb{E}[Z_t(\sigma)] \geq q_d$. □

Proof of Proposition 1(b). Let σ have i at position k and j at $k{+}1$, and let σ' swap them. Terms at positions $< k$ are identical. Terms at positions $> k{+}1$ carry the factor $\mathbf{1}_{F_{<k}}(1 - z_i)(1 - z_j)$ in both chains and are identical. Hence

$$C_t(\sigma) - C_t(\sigma') = \mathbf{1}_{F_{<k}}\big[c_i + c_j(1 - z_i) - c_j - c_i(1 - z_j)\big] = \mathbf{1}_{F_{<k}}\big[c_i z_j - c_j z_i\big].$$

Taking expectations and using the assumption that, given $F_{<k}$, c_i is independent of z_j and c_j of z_i,

$$\mathbb{E}[C_t(\sigma) - C_t(\sigma')] = \Pr[F_{<k}]\big(\bar{c}_i^{(k)} \rho_j^{(k)} - \bar{c}_j^{(k)} \rho_i^{(k)}\big),$$

which is ≤ 0 iff $\bar{c}_i^{(k)} \rho_j^{(k)} \leq \bar{c}_j^{(k)} \rho_i^{(k)}$. When both conditional success rates are positive, this is $\bar{c}_i^{(k)} / \rho_i^{(k)} \leq \bar{c}_j^{(k)} / \rho_j^{(k)}$. □

Proof of Proposition 1(c). Under independence across languages and of costs from other languages' outcomes, $\bar{c}_\ell^{(k)} = \bar{c}_\ell$ and $\rho_\ell^{(k)} = p_\ell$ for every prefix. Condition (b) then compares the fixed quantities $\bar{c}_\ell / p_\ell$, so any order that is not sorted by this index contains an adjacent inversion whose swap does not increase expected cost. Repeated swaps reach the sorted order without increasing cost, so the sorted order is optimal (Simon & Kadane, 1975). □

Break-even condition. For the chain (ℓ, rest) versus the default d alone, conditioning on the first attempt gives $\mathbb{E}[C_t(\ell, \text{rest})] = \bar{c}_\ell + \Pr[z_{t\ell}{=}0]\,\mathbb{E}[C_t(\text{rest}) \mid z_{t\ell}{=}0]$, and the chain is cheaper in expectation iff this is below $\bar{c}_d$. In REP, with $\ell =$ Ruby and rest = (Clojure, Python), $\bar{c}_\ell = 51.2$, $\Pr[z_{t\ell}{=}0] = 0.195$, and $\mathbb{E}[C_t(\text{rest}) \mid z_{t\ell}{=}0] = 242.7$, giving $98.5 < 161.7$. The chain would stop paying off only if the expected cost after a Ruby failure exceeded $(161.7 - 51.2)/0.195 \approx 567$ tokens. On McEval, where Ruby fails at the first attempt on 66% of tasks, the same chain costs 255.8 tokens against 279.6, close to break-even (Table 5).

Why a per-attempt bandit does not target Eq. (1). The bandit's reward for attempt (t, ℓ) is $(1 - \tau(c_{t\ell}))z_{t\ell}$; each head estimates the marginal expected reward of its language given task features, and a failed attempt updates only its own head with reward 0. Proposition 1 shows that the chain objective depends on quantities conditional on the failure of earlier languages, which such heads do not model; in addition, the clipped normalization τ saturates, so reward is not proportional to counted tokens. Choosing the language with the highest marginal reward is therefore neither necessary nor sufficient for a low charged cost.

B MULTILANG-BENCH CONSTRUCTION AND AUDITS

Tasks and splits. MultiLang-Bench contains 3,000 deterministic pure-function tasks, each with one accepted solution in C, Clojure, Go, Java, Julia, Python, Ruby, and Rust (24,000 solutions). The source mix is 990 HumanEval/MBPP/APPS-style tasks, 1,230 Codeforces/LeetCode-style tasks (input/output problems converted to pure functions), and 780 manually written or author-reviewed tasks. Prompts are language-neutral rewrites that keep a source reference rather than redistributing every upstream statement verbatim. The split is 2,100/450/450 train/validation/test and is fixed by seed. For each task, the manifest records a source identifier or note, prompt policy, license field, normalization note, domain and difficulty labels, argument/return type family, split, and deterministic tests. For each solution, it records the language, construction path (translation, curation, or author review), harness result, language-fidelity result, and token, character, and byte counts. A solution

is admitted only if it passes the full harness and the language-fidelity check, so "verified" means accepted by this harness, not proved correct.

Contamination audit. We remove exact, near-duplicate, and semantic overlaps between the training split and external benchmarks before evaluation, using SHA-256 hashes of normalized descriptions (lowercased, punctuation stripped), MinHash LSH on character 5-grams (Jaccard ≥ 0.8), and CodeBERT cosine similarity ≥ 0.95. Table 6 lists the matches; the two exact McEval matches come from the HumanEval/MBPP source pool and are excluded from training.

Table 6: Overlap between the MultiLang-Bench training split and external benchmarks (removed before evaluation).

Benchmark	n	Exact	Near	Semantic
HumanEval-X	164	0	3 (1.8%)	7 (4.3%)
MultiPL-E	1840	0	12 (0.7%)	31 (1.7%)
McEval	1000	2	8 (0.8%)	19 (1.9%)

Token variation. Across the corpus, the most and least expensive verified function bodies of a task differ by $2.19\times$ on average (95% CI $[2.17, 2.20]$). Table 7 gives per-language means, and Figure 4 gives medians for six 300-task families, whose cheapest language varies between Go and Python. Corpus statistics are counted with tiktoken's GPT-4o encoding. The corpus means are close for Python, Ruby, Julia, and Go (43.9–46.0 tokens): the large live differences in §5.2 come from the generator's language-conditional style, not from the reference solutions.

Table 7: Mean function-body tokens and characters per language over the 24,000 verified solutions.

Language	Tokens	Chars	Language	Tokens	Chars
Python	43.9	105.6	Java	47.4	120.8
Ruby	45.3	98.3	C	49.4	117.1
Julia	45.9	112.8	Rust	51.5	118.2
Go	46.0	115.2	Clojure	59.2	143.1

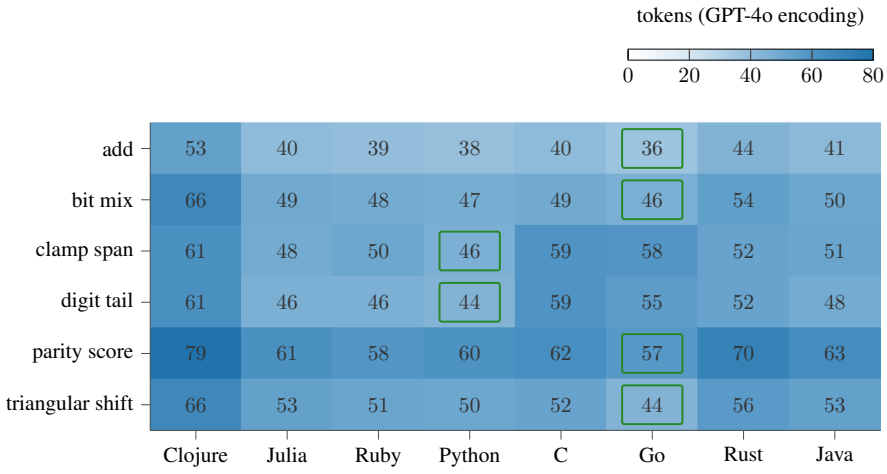


Figure 4: Median function-body tokens (GPT-4o encoding) of verified solutions for six 300-task families of MultiLang-Bench; rows are task families and columns are languages.

Scope. Requiring one accepted solution per language biases the corpus toward tasks that are solvable in all eight languages under a shared harness. The corpus is designed to test language-flexible routing; it does not represent repository-pinned software engineering.

C Evaluation Protocol

Executors and isolation. Executors are fixed across policies. Python uses `pytest`; Java uses JUnit 5 with a `Solution` class wrapper; Go uses `go test`; Rust uses `cargo test` with a `#[cfg(test)]` module; C uses `gcc` with an `assert.h` runner; Ruby uses `minitest`; Julia uses `Test.jl`; Clojure uses `clojure.test`. Each attempt runs in a fresh temporary directory with the same timeout and resource limits across languages; the harness does not use OS-level sandboxing. Compilation errors, runtime exceptions, timeouts, failed tests, and language-fidelity failures all count as failed attempts.

Language fidelity. The fidelity check combines parsing or compiling in the intended language's wrapper, validation of the expected entry point, and executor compatibility; missing or off-target code is a fidelity failure. The check is conservative: mixed-language comments or misleading identifiers can survive it. Before the partial-supervision campaign (SUP), we found and fixed two entry-point renaming defects in the harness: Clojure `defn` forms were not renamed to the expected entry point, and Julia's one-line assignment form was not matched. Campaigns that predate the fix may undercount correct Clojure and Julia attempts, which biases results against chains that use these languages (Appendix J).

Decoding and budget. All policies share one prompt template per language (the template asks for a function with the given entry point and no comments or docstrings), the same generator settings, and a budget of $B{=}3$ attempts. Every failed, fallback, and successful attempt is charged.

Accounting boundaries. The *harness-proxy* count is the number of tokens in the full emitted code string. The logs also contain *function+signature* and *function-body* counts, produced by a language-specific extractor. In the live REP logs this extractor is unreliable: it returns zero body tokens for 592 of 827 Ruby attempts, and it returns the full string for almost all Go, C, Clojure, Julia, Rust, and Java attempts. Live body-level savings (e.g., 63.6% for the fixed order in MAIN and 60.6% in REP) therefore overstate the saving, and we do not use them. Corpus-level counts do not show this pathology (mean Ruby body length 45.3 tokens; Appendix B). Function+signature counts track the harness proxy closely (MAIN: 49.7% vs. 50.3% saving).

Tokenizers. Live-campaign counts use tiktoken's `cl100k_base` encoding for every language and policy; corpus statistics (Appendix B, Figure 1) use tiktoken's GPT-4o encoding. Neither is the billing tokenizer of GPT-5 or DeepSeek-Coder, and neither counts hidden reasoning tokens. All comparisons are relative and within one encoding.

Statistics. Wilson intervals use $z = 1.96$. Bootstrap intervals resample tasks with replacement (10,000 resamples, fixed seed) and recompute the ratio $1 - \sum_t C_t(\pi) / \sum_t C_t(d)$ on each resample, keeping each task's policy and baseline costs paired. McNemar tests are exact two-sided binomial tests on the discordant pairs.

Python cache. Python generations are cached per task and reused by every policy that requests Python on that task. In the REP logs, all Python attempts inside bandit chains were served from this cache, and the task-level Python draw matches the Python-only policy on 400 of 400 tasks for the verifier outcome and on 399 of 400 for the token count. In HARD, the supervised selector's 59 Python attempts all match the cached draw.

Recomposition. For a chain σ' that uses only languages attempted by the logged fixed-order chain on a task, plus Python, we compute $C_t(\sigma')$ and $Z_t(\sigma')$ from Eq. (1) using the logged outcomes and token counts of those attempts and the task's cached Python draw. Because the fixed-order chain's logs contain Ruby for every task, Clojure for every task on which Ruby failed, and Julia for every task on which both failed, every chain in Table 4 is computable on all 400 tasks without imputation. The estimate is exact for the logged draws. Two assumptions separate it from a fresh run: that a Python attempt placed after failures would be served from the cache (as it is for the bandit policies), and that sampling variation in a fresh run matches that of the logged draws.

D Selector Details

Static and domain orders. The live fixed order is one global ranking, fixed before the live campaigns and applied to every task; in REP and HARD it tries Ruby, then Clojure, then Julia. This order

coincides with ranking the non-Python languages by how often each is the cheapest verified language under the stored harness-proxy training labels of the 377 warm-start tasks (Ruby 136, Python 125, Clojure 65, Julia 38, Go 9, Rust 2, C 1, Java 1). The domain order applies the same rule within each task domain; it is used only in replay (Table 9).

Supervised selectors. The supervised selector concatenates a frozen 768-dimensional CodeBERT embedding (Feng et al., 2020) of the task specification with one-hot metadata (source group, domain, difficulty, argument/return type family) and fits a linear classifier on the training split to predict the cheapest verified language; validation selects regularization. Its class scores order the whole chain. The six supervised baselines in Table 10 use the same features.

LinUCB bandit (hybrid prior with Neural-LinUCB heads). The CodeBERT embedding $e_t \in \mathbb{R}^{768}$ passes through an MLP ($768 \to 512 \to 256$, ReLU, dropout 0.1) to give $x_t \in \mathbb{R}^{256}$. The MLP is trained on the training split (Adam, learning rate 10^{-3}, 50 epochs, cross-entropy over cheapest-language labels) and then frozen. Each language keeps a design matrix A_ℓ and vector b_ℓ, initialized by warm start as $A_\ell^{(0)} = I + \sum_{i:\ell_i^*=\ell} x_i x_i^\top$ and $b_\ell^{(0)} = \sum_{i:\ell_i^*=\ell} x_i$. The score is

$$u(\ell) = \hat{\theta}_\ell^\top x_t + \alpha_n \sqrt{x_t^\top A_\ell^{-1} x_t}, \qquad \hat{\theta}_\ell = A_\ell^{-1} b_\ell, \qquad \alpha_n = 0.3/\sqrt{n}, \tag{4}$$

and after each attempt only the attempted language's head is updated, $A_\ell \leftarrow A_\ell + x_t x_t^\top$, $b_\ell \leftarrow b_\ell + r\, x_t$, with $r = (1 - \tau(c))\mathbf{1}[\text{pass}]$ and $\tau(c) = \text{clip}((c - c_5)/(c_{95} - c_5), 0, 1)$ using warm-start percentiles. A failed attempt updates its own head with $r = 0$; a later success is credited only to the language that passed. The bandit starts from the static prior and lets the UCB score override it. Languages absent from a task's warm-start record leave that head untouched, so censored logs need no special handling (Appendix H).

Parallel Python candidate and sequential cost. As implemented, the bandit policies generate a Python candidate alongside the bandit's pick on every task and add its tokens to the task's total, but record it as an attempt only if the fallback loop reaches it. The pre-charged Python tokens average 136.2 per task for the full bandit (nonzero on 262 of 400 REP tasks), 115.4 without warm start (235 tasks), and 161.7 without task features (all 400 tasks). The sequential cost sums only recorded attempts; it is what a sequential implementation with the same outcomes would pay.

Observed first choices. In REP the fixed order chose Ruby on 400/400 tasks; the bandit chose Go 252, Python 122, Clojure 24, Rust 1, C 1; without warm start it chose Julia 203, Python 137, Clojure 59, Ruby 1; without task features it chose C on all 400 tasks. In HARD, supervised selector chose Python 53 and Ruby 7; the bandit chose Rust 21, Go 17, Clojure 13, Java 6, and C, Julia, Python once each; its no-fallback variant chose Rust 34, Go 18, Clojure 3, and five other languages once each.

Idealized regret statement. For completeness we state the guarantee that motivated the bandit; it concerns attempt-level reward, not the charged chain cost of Eq. (1).

Assumption 1 (Attempt-level linear reward)**.** Each attempt n has context x_n with $\|x_n\|_2 \le 1$, chosen language ℓ_n from the set $\mathcal{L}_n$ of languages not yet tried on the task, and reward $r_n \in [0, 1]$ with $\mathbb{E}[r_n \mid x_n, \ell] = x_n^\top \theta_\ell^\star$, $\|\theta_\ell^\star\|_2 \le S$, and conditionally R-sub-Gaussian noise.

Theorem 1 (Attempt-level LinUCB regret; standard)**.** *Under Assumption 1, LinUCB with regularization $\lambda > 0$ and a confidence radius β_N that contains every $\theta_\ell^\star$ in the self-normalized confidence set with probability at least $1 - \delta$ has attempt-level reward regret $R_N = \sum_{n \le N} x_n^\top (\theta_{\ell_n^\star}^\star - \theta_{\ell_n}^\star) \le 2\beta_N \sqrt{2KN \log(1 + N/(K\lambda d))} = \tilde{O}(d\sqrt{KN})$ with probability at least $1 - \delta$, where $K = |\mathcal{L}|$.*

The proof is the standard LinUCB argument (Li et al., 2010) applied to the attempt stream and the shrinking action sets $\mathcal{L}_n$. The implemented policy uses a frozen neural representation and a tuned exploration schedule, so the theorem describes idealized update semantics, not the finite-sample behavior of the implementation, and Section 5.3 shows that low reward regret does not imply a low token count. Figure 5 shows the regret diagnostic from a 204-task adaptive trace, where the warm-started Neural-LinUCB accumulates less cost regret than plain LinUCB, ε-greedy, and random selection.

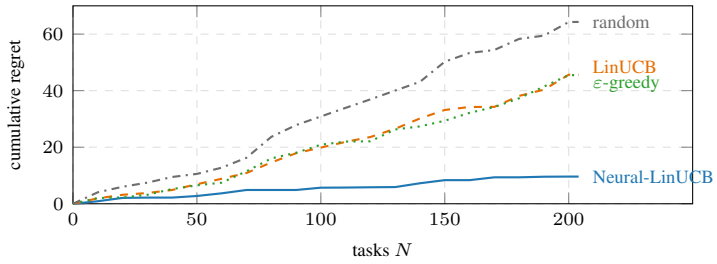


Figure 5: Cumulative cost-regret diagnostic on a 204-task adaptive trace. The trace is small and serves only as a mechanism check.

E Replay Results and Deployment Constraints

Held-out replay. Figure 6 shows replay on the 450-task test split under function-body accounting. Each policy's first language is assigned the length of that language's verified solution, so pass is 100% by construction and a wrong choice costs only extra length.

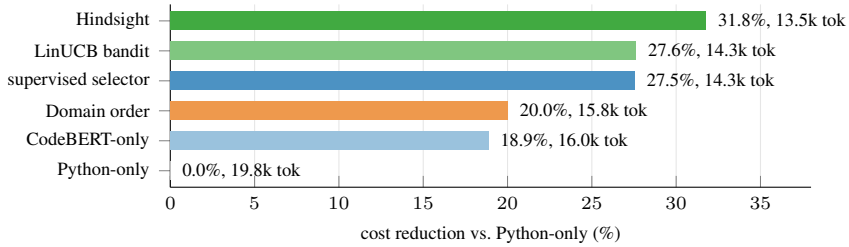


Figure 6: Replay on the 450-task MultiLang-Bench test split (function-body tokens; pass 100% by construction). Labels give the reduction and total tokens.

External benchmarks. In replay, the LinUCB bandit reduces function-body cost relative to Python-only by 28.3% on HumanEval-X, 31.6% on MultiPL-E, and 34.2% on McEval (all $p < 0.001$ after Bonferroni correction).

Workload partitions. Table 8 gives hindsight headroom by partition of the test split; the source-group rows reproduce the overall 31.8% when weighted by size.

Table 8: Hindsight replay headroom by partition of the test split (function body). Dominant: most frequent hindsight language.

Group	Tasks	Headroom	Dominant
Domain			
Math	58	41.21%	Clojure
String	60	41.47%	Ruby
Data structure	99	31.39%	Java
Numeric	57	8.78%	Python
Source group			
HumanEval/MBPP/APPS	136	38.11%	Ruby
Codeforces/LeetCode	203	26.31%	Java
Manual	111	34.25%	Ruby
Token-spread quartile			
Q1 (lowest spread)	127	22.94%	Rust
Q4 (highest spread)	112	39.19%	Ruby

Domain order versus the supervised selector in replay. Table 9 sums function-body tokens per domain over the 373 test tasks with all eight languages; the domain order uses each domain's training-split cheapest language.

Table 9: Per-domain replay cost (sum of function-body tokens; lower is better), n=373.

Domain	n	Train / test mode	Domain order	Supervised	Hindsight
Data structure	99	Go / Java	4,179	3,371	3,150
Graph	57	Rust / Rust	2,190	2,081	2,021
Math	58	Clojure / Clojure	2,120	1,691	1,531
Numeric	57	Python / Python	1,708	1,622	1,558
String	60	Ruby / Ruby	1,828	1,808	1,565
Systems	42	C / C	1,463	1,402	1,341
Total	373		13,488	11,975	11,166

Supervised baselines. Table 10 compares six supervised classifiers on a small held-out slice of the warm-start corpus (n=39); intervals are wide at this size. Random forest recovers the hindsight-optimal first language most often (61.5%) and saves 5.83%.

Table 10: Supervised routing baselines (replay, n=39, CodeBERT+metadata features, Wilson 95% CIs).

Baseline	First-language acc.	Oracle recovery	Cost red.	Replay red.
Random forest	46.2% [31.6, 61.4]	61.5% [45.9, 75.1]	+5.83%	+7.00%
Pairwise classifier	51.3% [36.2, 66.1]	56.4% [40.9, 70.7]	+4.06%	+2.40%
Logistic regression	46.2% [31.6, 61.4]	53.8% [38.6, 68.4]	+2.46%	+1.09%
Cost-weighted logistic	38.5% [24.9, 54.1]	53.8% [38.6, 68.4]	−1.44%	+1.60%
MLP	28.2% [16.5, 43.8]	46.2% [31.6, 61.4]	+2.66%	+2.91%
Gradient-boosted trees	25.6% [14.5, 41.1]	46.2% [31.6, 61.4]	−2.69%	+0.61%

Label boundary. Cheapest-language labels depend on the accounting boundary and on how it is measured. An earlier relabeling of the warm-start corpus rescaled its costs with per-language body/harness ratios estimated from 2,073 passing live attempts. Because the live body extractor is unreliable for Ruby (Appendix C), that relabeling made Ruby the cheapest language on 353/377 tasks (93.6%) under function-body accounting, against 112/377 (29.7%) under harness-proxy accounting, where Python rose to 170/377 (45.1%). The stored harness-proxy training labels instead have Ruby 136 and Python 125. We therefore train and report on harness-proxy labels only, and we read the body-level relabeling as a warning about boundary artifacts rather than as a result.

Deployment constraints. A deployment can restrict chains with a hard mask $A_t \subseteq \mathcal{L}$ and subtract soft penalties from any score:

$$\text{score}(\ell) = u(\ell) + \lambda_q \, \hat{q}(s_t, \ell) - \lambda_m M(\ell) - \lambda_i I(t, \ell), \tag{5}$$

where u is the policy's base score, $\hat{q}$ a pass-probability estimate, $M(\ell) \in [0, 1]$ an organization-level maintainability penalty (defaults: Python 0.0, Java 0.1, Go 0.1, Ruby 0.2, Rust 0.3, C 0.4, Julia 0.5, Clojure 0.6), and $I(t, \ell) \in [0, 1]$ a task-level integration penalty. Table 11 reports harness-proxy hindsight headroom on the warm-start corpus under common constraints. Dropping C and Rust costs 0.1 pp; a mainstream portfolio keeps 15.15%; soft penalties leave the headroom unchanged because the hindsight languages are rarely penalized.

Invariance checks. The cheapest-language labels are unchanged under seven monotone reward normalizations (percentile ranges p5–p95, p1–p99, p10–p90; log, raw, character, and byte scaling), and hindsight headroom on the warm-start corpus stays at 19.1%. Because these transformations are monotone, the invariance holds by construction; it shows only that the reward scale does not alter the labels, not that the result is robust to a different tokenizer.

Table 11: Hindsight replay headroom under portfolio constraints (n=377 warm-start tasks, harness proxy).

Scenario	Allowed languages	λ_m	Headroom
Full portfolio	all 8	0.0	19.09%
No low-level languages	all but C, Rust	0.0	19.00%
Mainstream + Rust	Python, Java, Go, Ruby, Rust	0.0	15.32%
Mainstream	Python, Java, Go, Ruby	0.0	15.15%
Soft penalty	all 8	0.3	19.09%
Soft penalty, mainstream	Python, Java, Go, Ruby	0.3	15.15%
Strong soft penalty	all 8	1.0	19.09%

F Additional Live Results

MAIN in full. Table 12 reports all MAIN rows under the three accounting boundaries. The hindsight row supplies only the first language; its attempt profile matches the fixed order (§5.2).

Table 12: MAIN (n=450, GPT-5): pass, fallback, attempts, selector latency, and tokens per task under three boundaries (reduction vs. Python-only in parentheses). Body counts come from an unreliable extractor (Appendix C) and are shown only for completeness.

Policy	First	FB rate	Final	Att.	SelLat (ms)	Body	Func.+sig.	Harness proxy
Python-only	92.0	0.0	92.0	1.00	0.01	130.7	139.5	141.4
Fixed order	87.3	12.4	92.9	1.20	0.01	47.6 (63.6%)	70.2 (49.7%)	70.3 (50.3%)
Supervised selector	91.8	8.0	93.8	1.14	0.94	132.7 (−1.5%)	145.0 (−3.9%)	146.7 (−3.7%)
Hindsight (first language only)	87.1	12.7	92.7	1.20	0.01	47.9 (63.4%)	70.8 (49.2%)	71.2 (49.6%)

REP in full. Table 13 lists all REP policies as run. The hindsight row had no per-task oracle labels in this campaign and therefore used the fixed order; its small differences from the fixed-order row reflect independent sampling of the same chain, which gives a sense of run-to-run noise (0.25 pp pass, 0.6 tokens per task).

Table 13: REP (n=400, GPT-5) as run (tokens per task, all attempts counted; tok/solved uses harness proxy; body counts are unreliable, Appendix C).

Policy	First	FB rate	Final	Att.	Body	Func.+sig.	Harness proxy	Tok/solved
Python-only	91.0	0.0	91.0	1.00	149.9	159.4	161.7	177.7
Fixed order	80.5	19.5	88.3	1.31	59.0	83.4	83.7	94.9
Hindsight (= fixed order here)	80.0	20.0	88.0	1.32	60.5	83.9	84.3	95.8
LinUCB bandit	76.3	23.8	85.0	1.43	308.4	317.9	320.3	376.8
without warm start	73.5	26.5	90.3	1.40	226.3	237.6	241.1	267.1
without task features	45.0	55.0	77.3	1.82	498.4	508.1	511.4	662.0

Per-language attempt statistics. Table 14 pools all REP attempts. The pooled pass rates mix first attempts and fallbacks, so languages tried mostly as fallbacks (Julia, Clojure) look worse than their first-attempt rates; the conditional rates in Table 15 are the relevant quantities for chain design. Passing Python attempts are 2.7× longer than passing Ruby attempts (120.4 vs. 44.4 tokens).

Table 14: REP attempt statistics per language (all policies; harness-proxy tokens).

	All attempts			First attempts only		
Language	n	Pass	Tok (passing)	n	Pass	Tok
Ruby	827	78.5	44.4	801	80.1	51.5
Python	704	88.9	120.4	659	91.0	124.1
Go	486	61.1	99.8	252	71.8	108.1
C	458	39.7	172.2	401	44.9	261.9
Julia	432	35.4	60.1	203	62.6	72.4
Clojure	340	49.4	82.7	83	66.3	72.6
Rust	45	4.4	48.5	1	0.0	109.0
Java	21	9.5	71.5	0	–	–

Table 15: Conditional cost and success along the fixed-order chain in REP. $\bar{c}$: mean harness-proxy tokens of the attempt; ρ: pass rate given that earlier positions failed; the Python column uses the cached Python draw of the same tasks.

Position	Tasks	**Candidate**	$\bar{c}$	ρ	$\bar{c}/\rho$	**Python**	$\bar{c}$	ρ	$\bar{c}/\rho$
1 (all tasks)	400	Ruby	51.2	0.805	63.6	Python	161.7	0.910	177.7
2 (Ruby failed)	78	Clojure	114.1	0.397	287.2	Python	275.0	0.603	456.3
3 (Ruby, Clojure failed)	47	Julia	87.2	0.000	∞	Python	213.4	0.468	455.8

Case studies (HARD). Table 16 lists every HARD task solved by exactly one of the fixed order and the supervised selector. In the six tasks that only the supervised selector solves, all three static-chain languages fail and the supervised selector's Python passes, at 187–701 tokens. On the 44 tasks both solve, the fixed order is cheaper on 36; for example, `truncate_number` costs 12 tokens in Ruby and 90 in Python, and `add` costs 10 and 69.

Table 16: HARD tasks solved by exactly one of the fixed order and the supervised selector: attempts as (language, pass, harness-proxy tokens).

Task (McEval)	**Fixed order**	**Supervised**
`optimize_seating`	Ruby F 198, Clojure F 265, Julia F 281	Python P 701
`count_passing_ways`	Ruby F 56, Clojure F 111, Julia F 37	Python P 290
`count_partitions`	Ruby F 121, Clojure F 107, Julia F 74	Python P 187
`count_sequences`	Ruby F 46, Clojure F 80, Julia F 47	Python P 303
`count_coprime_pairs`	Ruby F 85, Clojure F 171, Julia F 0	Python P 455
`preorder_traversal`	Ruby F 134, Clojure F 197, Julia F 156	Ruby F 133, Python P 271
`find_topmost_carpet`	Ruby F 56, Clojure P 78	Python F 90, Ruby F 62, C F 150

Selector latency. Table 17 decomposes selector time in a CPU microbenchmark (n=1,000 per stage; stages timed independently, so shares need not sum exactly). In the live REP logs, the embedding step took 243 ms per task on average (median 218 ms, p95 563 ms), and the policy's scoring step 0.44 ms; the fixed order needs neither.

Table 17: Selector latency decomposition (CPU microbenchmark, ms; the metadata mean is inflated by rare outliers, its median is 0.015 ms).

Stage	**Mean**	**p95**	**Share**
Metadata extraction	0.354	0.026	0.9%
CodeBERT embedding	39.361	50.277	99.3%
LinUCB scoring	0.291	0.350	0.7%
Total selector	39.65	50.63	100%

Combining with model routing. Target-language routing and model routing act on different variables. If their savings r_ℓ and r_m were independent, the combined saving would be $1-(1-r_\ell)(1-r_m)$; with $r_\ell = 0.39$ (Ruby→Clojure→Python, REP) and an illustrative $r_m = 0.4$, this gives about 0.63. We have not run a joint experiment, and a cheaper model may write non-Python languages less reliably (§5.5), so this is an accounting identity rather than a result.

G CONDITIONS: PROMPT STYLE, GENERATOR, AND HIDDEN COSTS

Prompt-style control (STYLE). STYLE generates three Python variants per task on 450 held-out MultiLang-Bench tasks (three buckets of 150: algorithmic, competitive, and curated): the default prompt; a terse prompt, which repeats the default instruction to omit comments and docstrings and adds instructions to prefer comprehensions and built-ins and to use short conventional names; and the terse output passed through an AST pass that strips remaining comments, docstrings, and type annotations and renames locals. Table 18 reports all three boundaries. Savings are consistent across buckets (31.9–50.1%) and boundaries (41.6–44.5% in aggregate), and compaction saves few additional tokens at a clear cost in pass rate. The default-prompt pass rate in STYLE (68%) is lower than that of Python-only in MAIN (92%), so we compare the three variants only with one another.

Table 18: Prompt-style control (STYLE, n=450, GPT-5, Python only). Tokens per task; saving relative to the default prompt; single-attempt pass rate.

	Default	Terse prompt		Terse + AST compaction	
Boundary / bucket	Tok	Tok	Saving	Tok	Saving
Function body (all)	130.3	72.3	44.5%	72.3	44.5%
Function + signature (all)	142.1	82.2	42.1%	79.8	43.8%
Harness proxy (all)	145.9	85.1	41.6%	82.6	43.4%
Harness proxy, algorithmic (pass 89/89/76%)	141.0	78.3	44.5%	71.6	49.3%
Harness proxy, competitive (pass 63/55/43%)	154.6	101.4	34.4%	105.3	31.9%
Harness proxy, curated (pass 53/51/47%)	141.9	75.7	46.7%	70.9	50.1%
Pass rate (all)	68%	65%		55%	

Cross-campaign composition. If routing saves s_r and style control saves s_p relative to the same default-Python cost, routing saves $1-(1-s_r)/(1-s_p)$ relative to terse Python. With $s_p = 0.416$, this gives 14.9% for the MAIN fixed order ($s_r = 0.503$), 11.3% for the REP fixed order ($s_r = 0.482$), and -4.3% for Ruby→Clojure→Python ($s_r = 0.391$). These numbers combine campaigns with different task mixes and pass rates, so they indicate magnitude only. The direct test, routing and terse prompting in every language on the same tasks, remains to be run.

Open-weight generator (OPEN). OPEN runs DeepSeek-Coder-6.7B-Instruct, served locally with vLLM on one A10G GPU, on 20 tasks from each of HumanEval-X, MultiPL-E, McEval, and MultiLang-Bench. Each task gets a bandit-selected language and a Python candidate generated in parallel, and the evaluator returns a passing candidate. Its per-task cost field records only the returned program. We therefore recount all attempts: 600.4 tokens per task as run (both candidates always generated) and 592.0 for the sequential equivalent in which Python is generated only after the bandit's attempt fails (Table 19). The bandit's own choices pass on 9 of 80 tasks (Rust 6/45, Ruby 3/17, Go 0/11, Java 0/4, Clojure 0/1, Julia 0/1, C 0/1). Sampled failures contain untranslated Python syntax (e.g., `def` signatures with `typing` imports) inside code requested in another language. A second run with exploration $\alpha = 0.05$ gives the same picture (bandit 2/80; routed 47/80 vs. Python 45/80; 553.3 vs. 288.9 tokens with all attempts counted). A partial live-evaluator run on DeepSeek (6 HumanEval-X tasks) shows Ruby 0/12, Julia 0/11, and Clojure 2/13 attempts passing versus Python 16/16; a 40-task run of the same evaluator failed to produce any generations and is excluded.

Table 19: DeepSeek-Coder-6.7B (OPEN, 20 tasks per benchmark). Routed = bandit choice with Python fallback. Winner-only counts only the returned program; all attempts counts every attempt (Eq. (1)).

Benchmark	**Routed pass**	**Python pass**	**Routed, winner-only**	**Routed, all attempts (seq.)**	**Python**
HumanEval-X	19/20	19/20	257.7	529.9	135.2
MultiPL-E	14/20	14/20	184.4	531.6	191.4
McEval	4/20	1/20	291.6	756.0	475.5
MultiLang-Bench	12/20	12/20	212.1	550.5	224.8
Total	49/80 (61.3%)	46/80 (57.5%)	236.5	592.0	256.8

Wall-clock time. Table 20 compares generation time per task. The Python draws were generated earlier and cached, so their times come from a different period of API load; the comparison is indicative. In both campaigns the fixed-order chain takes longer than the Python draw, even though it emits far fewer code tokens.

Table 20: Generation wall-clock time per task (seconds, GPT-5).

Campaign	**Ruby first attempt** mean	**Python draw** mean	**Ruby slower** (share of tasks)	**Fixed-order chain total** mean / median	**Python draw** median
REP (n=400)	18.6	14.9	64.8%	27.9 / 15.9	9.5
HARD (n=60)	29.9	25.5	71.7%	57.7 / 38.3	13.6

H SUPERVISION REQUIREMENTS AND ROBUSTNESS

What deployment needs. Full per-language coverage (a verified solution in every language for every task) is needed only to compute hindsight references and replay headroom. A deployed policy observes only the languages it tries: for each attempt, the language, the verifier outcome, and the token count. The fixed order needs per-language cost and pass statistics, which such logs provide. The bandit's warm start updates only the heads of languages present in a task's record, so an untried language is treated as unknown rather than as a failure, and censored logs need no algorithmic change.

Per-language censoring (SUP). We generated and verified all eight languages with GPT-5 on 150 fresh MBPP and LeetCode tasks, disjoint from the original warm-start corpus and from the evaluation tasks. Per-language passes were Julia 120, Go 120, Ruby 113, Python 113, Clojure 106, Rust 104, Java 79, and C 73; the number of passing languages per task was 1 (11 tasks), 2 (7), 3 (10), 4 (8), 5 (21), 6 (42), 7 (24), or 8 (27). To simulate a budget of r attempted languages per task, we reveal r random languages per task (seed 42) and label each task with the cheapest passing revealed language, dropping tasks with none. We warm-start the bandit with each corpus and evaluate it live on the 400 REP tasks (Table 21). Censoring to two languages per task does not lower the final pass rate. Under the sequential accounting, all variants save 30–41% relative to the campaign's cached Python draw (163.7 tokens, 90.5% pass) but remain more expensive than the fixed order.

Table 21: LinUCB bandit warm-started with r of 8 languages revealed per task (SUP, n=400). As run includes the parallel Python candidate; Sequential counts only the attempts used.

Warm start	Labeled tasks	First	Final	FB rate	As run	Sequential
$r = 2$	126/150	78.0	89.5	22.0	256.0	114.2
$r = 4$	143/150	80.0	89.8	20.0	248.1	99.2
$r = 6$	144/150	79.8	89.8	20.3	250.5	96.8
$r = 8$	150/150	78.5	87.8	21.5	253.0	101.4
Fixed order (REP, same tasks)	–	80.5	88.3	19.5	83.7	83.7

Task-level coverage. In a separate study with a different evaluation set, we dropped whole tasks from the original 377-task warm-start corpus (Table 22; 236 tasks from McEval and MultiLang-Bench, cached Python draw 86.4% pass, 153.6 tokens). At 50% coverage, first-attempt pass drops to 55.9% and the final pass rate to 84.3%, although the trend is not monotone. Because the evaluation sets and warm-start corpora differ, this study cannot be compared directly with Table 21; a plausible reading is that fewer tasks give the heads fewer distinct contexts, whereas censoring languages leaves every context informative.

Table 22: LinUCB bandit warm-started on a fraction of warm-start tasks (n=236).

Coverage (tasks)	First	Final	FB rate	As run	Sequential
100% (377)	71.2	87.7	28.8	277.4	174.0
75% (283)	69.1	84.3	30.9	285.3	185.2
50% (188)	55.9	84.3	44.1	333.4	200.4
25% (94)	72.5	86.0	27.5	268.0	183.6

Label noise and domain shift (replay). We corrupt a fraction p of the cheapest-language labels of the 377 warm-start tasks (uniform relabeling; $p = 1$ permutes the label column), rebuild each policy, and score its single-shot first choice against the true per-task costs (harness proxy, no fallback). Here the global order is rebuilt by label frequency from the (possibly corrupted) labels of this experiment; with clean labels it is Ruby, Python, Julia, Clojure, so it includes Python and is close to, but not identical with, the live chain. Table 23 shows that the order changes little up to 50% noise (its first choice is Ruby, or Python at $p = 0.1$) and collapses only under full permutation (Java first), while the greedy bandit degrades earlier. Across source domains, a policy fit on MBPP and tested on LeetCode loses little as a fixed order (-2.1%) but much as a bandit (-33.4%, versus -14.9% in-domain). Single-shot scoring without fallback makes these reductions small or negative; they measure selection quality, not deployable savings.

Table 23: Label noise and domain shift (replay, single-shot first choice, harness proxy). Red.: cost reduction vs. Python-only; Match: first choice equals the true cheapest language.

		Global order		Bandit (greedy)	
Setting	**Hindsight**	Red.	Match	Red.	Match
Noise $p = 0$		3.3%	36.1%	−5.7%	28.4%
Noise $p = 0.1$		0.0%	33.2%	−21.4%	18.6%
Noise $p = 0.25$		3.3%	36.1%	−23.0%	27.1%
Noise $p = 0.5$		3.3%	36.1%	−51.1%	9.8%
Noise $p = 1$ (permuted)		−61.6%	0.3%	−23.7%	11.9%
LeetCode → MBPP (127)	26.5%	0.0%	22.1%	0.0%	22.1%
MBPP → LeetCode (250)	15.3%	−2.1%	38.4%	−33.4%	10.4%
In-domain 50/50 (189)	19.0%	0.0%	32.8%	−14.9%	19.1%

I Result Ledger

Table 24 maps each headline number to its campaign and log. File names refer to the per-attempt logs of each campaign.

Table 24: Provenance of headline numbers.

Number(s)	**Campaign**	**Source**
2.19× spread; per-language means; heatmap	corpus	corpus token-count report (GPT-4o encoding)
Replay 31.8/27.6/27.5/20.0/18.9%; partitions	R	replay traces on the test split (function body)
Replay 19.1% and portfolio constraints	R (warm-start, 377)	constrained-portfolio and deployment-constraint reports
External replay 28.3/31.6/34.2%	R (external)	replay on HumanEval-X, MultiPL-E, McEval
50.3%, 92.9 vs. 92.0%, 93.8%, −3.7%	Main (450)	Main live-evaluation log and summary (May 2026 run)
48.2% [40.9, 55.0], 88.3 vs. 91.0%, b/c	Rep (400)	Rep live-evaluation log (July 2026 run); CIs and tests recomputed from per-task records
Bandit ablations; waste share; conditional rates; recomposed chains; per-benchmark; wall-clock	Rep (400)	same file; as-run and sequential counts recomputed from per-attempt records
Per-task saving (347/400, median 57.1%); Python test-retest (394/400, r=0.81)	Rep, Sup (400)	Rep log and the Sup r=8 log, matched by task id
Supervised selector vs. bandit; no-fallback bandit; case studies	Hard (60)	`live_eval_xrst_ablation_results.json`
k-censoring and task coverage	Sup	`live_eval_k{2,4,6,8}.json`, `live_eval_cov{25,50,75,100}.json`
Label noise and domain shift	R (warm-start, 377)	`domain_robustness_results.json`
41.6%, 43.4% style savings	Style (450)	minimal-code baseline report
DeepSeek 61.3 vs. 57.5%, 592.0 vs. 256.8 tokens	Open (80)	`comprehensive_results_deepseek.json`; charged cost recomputed
Selector latency 39.65 ms	microbenchmark	selector latency report

Reconciliation notes. (i) Main and Rep differ in task mix (Main: MultiLang-Bench test split; Rep: four benchmarks), date, and sampling; the fixed order's saving agrees (50.3% vs. 48.2%), while its pass difference to Python-only is +0.9 pp in Main and −2.8 pp in Rep. (ii) In Rep, the hindsight policy had no per-task labels and reduces to the fixed order; we do not use it as an oracle in live results. (iii) The Rep benchmark counts are HumanEval-X 50, MultiPL-E 114, McEval 50, and MultiLang-Bench 186. (iv) The Open evaluator's cost field counts only the returned program; we report counts over all attempts. (v) Replay headroom on the test split is 31.8% (function body) and on the warm-start corpus 19.1% (harness proxy); these use different tasks and boundaries.

J Extended Limitations

Cost is emitted code, not the bill. We count tokens of emitted code with a public encoding. GPT-5 bills hidden reasoning tokens that our logs do not contain, each extra attempt pays the prompt again, and wall-clock time rose under routing (Table 20). A deployment that pays per reasoning token should measure the provider's reported usage before adopting a routing policy; our findings about chain design (Proposition 1) hold for any per-attempt cost, including billed tokens or latency, once that cost is measured.

Recomposition is not a fresh run. The Ruby→Clojure→Python chain is evaluated on logged draws from the campaign that suggested it. It is exact for those draws, and the absence of lost tasks relative to Python-only follows from reusing the Python draw (Proposition 1(a)), but its saving should be confirmed prospectively on fresh tasks. In a fresh run on 60 of these tasks (HARD), the saving was 13.2% with a wide interval.

Style is a confounder. Default GPT-5 Python is verbose; a terse prompt removes 41.6% of its tokens. Our routing savings are measured against the default prompt, and the net routing effect after style control is a cross-campaign estimate. Terse prompting in every language, combined with routing, is untested.

Generators. Live results cover GPT-5 and one 6.7B open-weight coder. Other frontier and open-weight models may differ in language-conditional verbosity and in how reliably they write each language, which Section 5.5 shows to be decisive.

Harness defects. Two entry-point renaming defects (Clojure `defn` forms; Julia one-line definitions) were fixed during SUP. Campaigns run before the fix may undercount correct Clojure and Julia attempts, which would penalize chains that use them.

Scope and validity. Tasks are function-level and polyglot-solvable; verification is by unit tests and a conservative language-fidelity check, which are guardrails rather than proofs. We do not measure maintainability, readability, or developer familiarity, which often decide language choice in practice.