CodeFORGE: Framework for Orchestrated Rare Programming Language Generation and Evaluation

Zhuoyan Yu*, Minghao Shao†‡, Yijun Shen*, Yichen Zhao*, Boyuan Chen†‡, Yik-Cheung Tam*, Muhammad Shafique‡
*Center for Data Science, NYU Shanghai, China †NYU Tandon School of Engineering, USA ‡NYU Abu Dhabi, UAE
{zy3077, shao.minghao, ys6126, yz10469, bc3194, wilson.tam, muhammad.shafique}@nyu.edu

***Abstract*—Large language models (LLMs) have shown strong code generation ability, yet most benchmarks concentrate on high-resource languages such as Python, Java, and C++, leaving a gap in evaluating LLM performance on low-resource and legacy languages. Languages including COBOL, Fortran, Ada, and Prolog remain in financial infrastructure, aerospace systems, scientific computing, and symbolic reasoning, but are supported by far less training data and lack widely accepted benchmarks. To fill this gap, we present `CodeFORGE`, an automated benchmark synthesis framework tailored to data-scarce programming languages. Our method uses LLMs to generate programming tasks together with corresponding test suites through a designed pipeline. `CodeFORGE` adopts a model-cascading verification strategy: multiple LLMs attempt each generated task in sequence, leveraging iterative feedback from earlier failures to enable cross-validation and ensure benchmark reliability. Only tasks that satisfy automated checks inside isolated Docker environments are included in the final benchmark. With this framework, we build `CodeFORGE-Bench`, a comprehensive benchmark spanning 12 low-resource languages across diverse paradigms, including procedural, functional, and logic programming. Extensive experiments evaluating state-of-the-art LLMs reveal performance gaps relative to high-resource languages, offering insights into current limitations when LLMs face linguistically diverse and data-scarce code generation settings.**

I. Introduction

Large language models (LLMs) have recently achieved strong results on program synthesis and competitive-programming-style tasks, benefiting from larger model scale and the abundance of training data for mainstream languages [1]. Progress is typically quantified through execution-based benchmarks such as HumanEval and related program synthesis suites, where a generated program is compiled when applicable and then executed against hidden tests [2]–[4]. Recent systems further push toward competition-level settings and large-scale sampling-and-filtering pipelines [5]. Despite these advances, current evaluation practice remains concentrated on a small set of high-resource languages, most notably Python, Java, and C++, which leaves a central question insufficiently answered: *how reliably do LLMs generalize to programming languages that are rare in training data and less supported by tooling?*

This gap is practical. Long-tail languages still underpin critical infrastructure and specialized workflows: from enterprise and finance to scientific/numerical computing and symbolic programming, yet public benchmarks and standardized evaluation harnesses for them are scarce. Without comparable and executable evaluation, it is difficult to quantify capability gaps and diagnose language-specific failure modes beyond mainstream ecosystems.

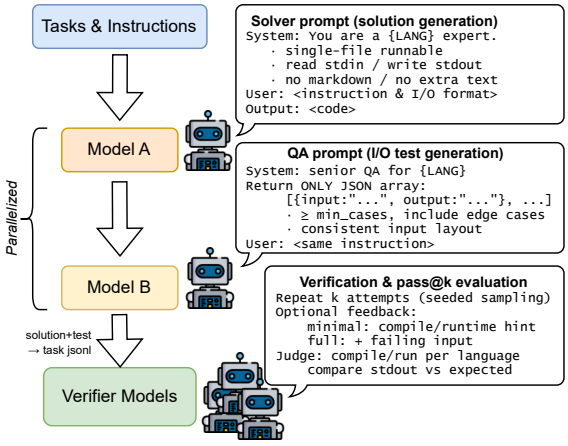

Fig. 1. A demonstration in `CodeFORGE` for solution generation (Model A), QA-style test generation (Model B), and verification in the pass@k loop.

Building such benchmarks manually is costly and brittle: each language requires its own compilation/runtime environment, entrypoint conventions, and I/O rules, and naive porting can introduce library or numerical mismatches. Moreover, weak tests can inflate pass@k and mis-rank models, so benchmark construction must include rigorous executable verification [6]. To address these challenges, we present `CodeFORGE`, an automated framework for synthesizing *executable* multi-language code-generation benchmarks. Rather than relying on manually written problem statements, `CodeFORGE` starts from a small set of task families (e.g., arrays, strings, arithmetic) and uses an LLM to synthesize many concrete, language-agnostic I/O instructions. For each language–instruction pair, we decouple **solution generation** (Model A) from **test generation** (Model B). Model A produces a complete single-file program that follows a strict stdin/stdout contract; Model B produces a JSON list of I/O tests without seeing the solution. We execute candidates inside a unified Docker environment with per-language compile/run commands, and admit only tasks that pass automated checks for parseable tests, deterministic behavior, and executable correctness.

A central ingredient is **model-cascading pass-k verifica-**

tion. Multiple verifier models attempt each task in sequence for up to k tries, receiving concise feedback between attempts (compiler diagnostics or the first failing input, without revealing expected outputs). A task enters the benchmark as soon as any verifier succeeds under the same harness; tasks that fail all verifiers are retained separately as unverified. This multi-round design reduces spurious rejections caused by any single model's language-specific blind spots while preserving a clear, reproducible admission rule.

Using `CodeFORGE`, we construct `CodeFORGE-Bench` [1], a fully executable benchmark spanning 12 low-resource and legacy languages (Ada, COBOL, D, Fortran, Julia, Lisp, Lua, OCaml, Octave, Prolog, REXX, Tcl). The benchmark is stored in a uniform JSONL format with per-task test suites, supports reproducible Docker-based evaluation, and is extendable to new languages and models with minimal engineering overhead. We evaluate seven state-of-the-art LLMs on `CodeFORGE-Bench` under a pass@5 protocol, revealing large and systematic performance disparities across languages. This enables systematic measurement of how model capability varies across programming ecosystems and supports diagnosis of failure patterns that are otherwise hidden when evaluation is confined to a narrow set of high-resource languages.

We summarize our main contributions as follows:

1) **Framework.** We introduce `CodeFORGE`, a pipeline that synthesizes executable multi-language code-generation benchmarks from task families and LLM-synthesized I/O specifications.
2) **Verification.** We propose feedback-driven pass-k verification with multi-round, model-cascading admission.
3) **Benchmark.** We release `CodeFORGE-Bench`, a curated 12-language benchmark with per-task I/O tests and a reproducible Docker harness.
4) **Empirical study.** We evaluate a diverse set of LLMs on `CodeFORGE-Bench`, surfacing long-tail language gaps and characteristic failure patterns.

II. Background and Related Works

A. Execution-based evaluation for code generation

A common setting for code generation benchmarks [1] is *I/O programming*: each task specifies an input format, a required computation, and an output format. A model is asked to generate a complete program that reads from standard input and writes the required output to standard output. AlphaCode similarly adopts this execution-based paradigm for competitive programming evaluation [5]. Correctness is then evaluated by compiling (when applicable) and executing the generated program against a hidden test suite [2], [3].

Because LLM outputs are stochastic, modern benchmarks often report *pass-k* metrics: a task is considered solved if *any* of k independently sampled programs passes all tests [2]. This paper adopts execution-based pass@5 (Section IV), which measures a model's probability of solving tasks when allowed up to five attempts under the same runtime harness. A recurring challenge is that test suites are often incomplete, meaning that apparently "passing" code can still be semantically incorrect; recent work (e.g., EvalPlus) emphasizes strengthening evaluation by augmenting tests and using more rigorous automated checking [6]–[8]. Moreover, benchmark contamination, where evaluation instances (or near-duplicates) appear in the pretraining corpus which can further inflate execution-based scores and complicate their interpretation [9].

[1]Our code is publicly available at: https://github.com/NYU-Serendipity-LLM4Coding/CodeFORGE.

B. Multi-language benchmarking and the low-resource gap

Most widely used code generation benchmarks focus on high-resource languages, which have abundant training data, mature tooling, and standardized evaluation environments [2]–[4], [10]. Recent efforts have expanded evaluation to multiple languages, notably via translation- or porting-based suites such as MultiPL-E and massively multilingual settings such as McEval and xCodeEval [11]–[14]. However, in practice these benchmarks still concentrate on mainstream ecosystems and only partially cover legacy or niche languages, leaving a blind spot: strong performance in Python or Java does not necessarily imply robust generalization to languages with different paradigms, stricter compilation constraints, or less common runtime conventions.

Low-resource and legacy languages introduce additional evaluation friction. Toolchains may be less standardized, build and execution conventions vary (e.g., entrypoint requirements, script vs. compiled workflows), and minor mismatches in I/O formatting can dominate measured performance even when underlying algorithmic reasoning is correct. Beyond evaluation harness issues, there is also evidence that cross-language transfer in programming languages is uneven and can depend strongly on syntactic and semantic distance, dataset composition, and training objectives [15]. These factors motivate benchmarks that are deliberately designed around low-resource language toolchains and uniform, reproducible execution.

C. Automated benchmark synthesis and verification

One scalable approach to benchmark construction is *benchmark synthesis*: generate candidate tasks, solutions, and tests using LLMs, then filter aggressively to retain only reliable, executable items. Such generation-and-filtering paradigms appear in both instruction data bootstrapping (e.g., Self-Instruct, WizardLM, ReST) [16]–[18] and in code-generation systems that rely on large-scale sampling plus behavioral filtering (e.g., AlphaCode, DOCE) [5], [19]. For executable programming benchmarks, synthesis pipelines face two recurring issues: (i) **executability noise** (syntax/compilation/runtime errors), and (ii) **self-consistency failures** (tests that do not match the intended specification, or solutions that fit some tests but not the true task).

`CodeFORGE` addresses these issues with a verification-centric pipeline built around two principles.

a) Separation of generation roles: We decouple solution generation from test generation by assigning them to two separate LLM calls. Model A produces a single-file canonical

solution that follows strict stdin/stdout conventions, while Model B produces a diverse I/O test suite based only on the natural language instruction, avoiding tight coupling between tests and a particular implementation.

b) Cross-validated, feedback-driven verification: Even with careful prompting, some candidates are flawed. CodeFORGE therefore verifies every task by executing programs inside a unified Docker environment and applying a pass-k solver procedure with iterative feedback: when an attempt fails to compile or fails a test, the verifier receives a concise failure signal (compiler error excerpt or a failing input) and retries, similar to recent self-debugging approaches that leverage execution feedback to iteratively refine generated code [20]. To reduce idiosyncratic false negatives from a single verifier, we run multiple verification rounds with different solver models in a cascade and admit a task if any verifier succeeds within budget.

Together, these design choices yield a benchmark whose tasks are both *executable* and *self-consistent* under a single, reproducible evaluation harness, enabling meaningful comparisons of LLM code generation capability across low-resource and legacy programming languages.

III. Methodology

Building executable benchmarks for low-resource languages poses unique challenges: each language requires its own compilation environment, entrypoint conventions, and reliable test cases, making manual curation prohibitively expensive. CodeFORGE addresses these challenges through an automated pipeline that generates candidate tasks via LLMs and filters them through feedback-driven, multi-model verification. The remainder of this section details each stage of this pipeline; evaluation is covered in Section IV.

A. Pipeline Overview

CodeFORGE builds executable benchmarks by synthesizing language-agnostic I/O task instructions from task families and transforming each language–instruction pair into a verified programming problem with a reference solution and comprehensive I/O tests. Central to our approach is a *model-cascading verification* strategy: multiple LLMs attempt each task in sequence, leveraging iterative feedback from earlier failures, and a task is admitted to the benchmark only if at least one model succeeds. This design filters out noisy or ambiguous candidates while avoiding spurious rejections caused by any single model. The pipeline is automated and readily extensible to new languages and models. The construction proceeds through four stages:

1) **Task family specification.** We define collections of programming problems organized by category, such as string manipulation, numeric computation, and data structures, and use an LLM to synthesize many concrete, language-agnostic I/O task instructions from these families.
2) **LLM-based candidate generation.** For each language–instruction pair, we prompt LLMs to generate a canonical solution and an I/O test suite. Crucially, solution generation and test generation are assigned to separate model calls to avoid coupling tests to implementation artifacts. Candidates are collected in an *incoming* pool.
3) **Feedback-driven verification.** Each candidate is compiled and executed in an isolated Docker environment. A sequence of LLM verifiers attempts to solve the task, each receiving feedback from previous failures. A task enters the *benchmark set* once any verifier succeeds within a fixed attempt budget; tasks that fail all verifiers are moved to an *unverified* pool.
4) **Model evaluation and difficulty bucketing.** We evaluate diverse LLMs on the verified benchmark and derive empirical difficulty labels based on cross-model success rates, as detailed in Section IV.

Figure 2 illustrates this pipeline. Each task is stored as a JSON object containing the language identifier, natural language instruction, canonical solution, I/O tests, and verification metadata. We maintain separate per-language files for the incoming, benchmark, and unverified splits in JSON Lines format, ensuring a uniform representation across all stages.

B. Programming Language Selection

CodeFORGE targets the long tail of languages underrepresented in standard LLM coding benchmarks. In this release, we include 12 rare/legacy languages: Ada, COBOL, D, Fortran, Julia, Lisp, Lua, OCaml, Octave, Prolog, REXX, and Tcl. The selection follows four criteria:

- **Complementarity.** We prioritize languages underrepresented in existing benchmarks such as HumanEval, MultiPL-E, and McEval [2], [11], [12], while ensuring diversity across paradigms and toolchains.
- **Real-world relevance.** We select languages with meaningful deployment in industry or research, including numerical computing, embedded systems, and symbolic reasoning, rather than purely academic or toy languages.
- **Tooling availability.** Each language must have a command-line compiler or interpreter installable in a non-interactive Docker environment, enabling fully automated execution.
- **Single-file execution.** Tasks must be solvable with a single source file and a simple compile-or-run command. Languages requiring complex build systems are excluded.

The resulting benchmark covers procedural, functional, and logic programming paradigms, both static and dynamic typing, and varying ecosystem maturity. While the initial languages set is fixed for our experiments, the pipeline can accommodate additional languages with minimal engineering effort.

C. Task Families and LLM-Synthesized Instructions

A core design goal is to scale task authoring without relying on human-written problem statements. Instead, we start from a small set of task families (topics) such as *array*, *string*, and *number theory*, and use an LLM to synthesize many concrete task instructions within each family. Each instruction is language-agnostic and explicitly specifies: (i) the input layout, (ii) the computation, and (iii) the output format. Example synthesized instructions include:

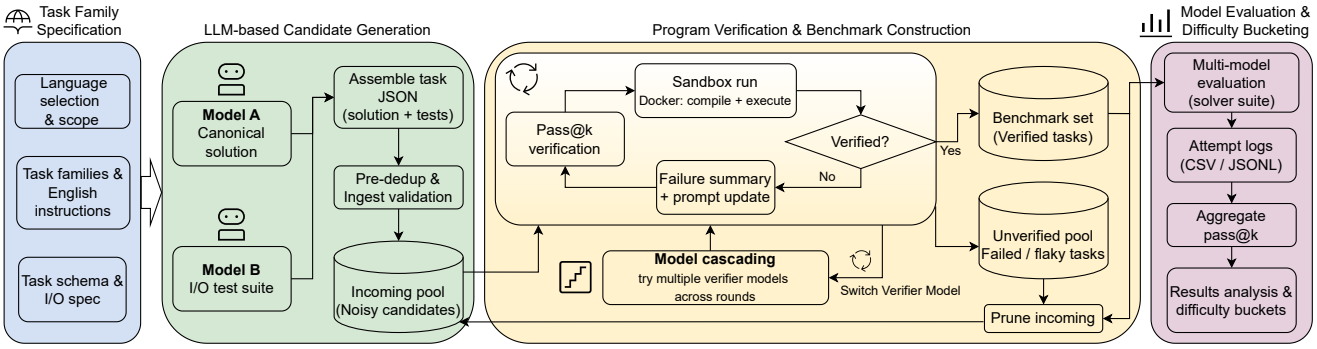

Fig. 2. Overview of our multi-language benchmark construction pipeline. Task families are expanded into language-agnostic I/O instructions and transformed into candidate tasks (incoming pool) via LLM-based generation, filtered through multi-round pass-k verification into the benchmark set, and then used for multi-model evaluation.

- *"Read n then n integers; print the 1-based index of the first occurrence of the maximum value."*
- *"Read n then n integers; print the length of the longest contiguous block where all numbers are equal."*
- *"Read n then n integers (which may be negative); print the maximum subarray sum over all contiguous subarrays."*

We emphasize that these instructions (and the associated short natural-language descriptions used for readability) are LLM-synthesized, not human-written. Manual additions are supported when needed (e.g., to cover underrepresented topics), but are not required for the pipeline to function.

D. LLM-based Task and Test Generation

Given the pool of task instructions and the set of target languages, we generate candidate tasks by querying LLMs in a batched and balanced manner. The generation process separates responsibilities across two distinct LLM calls, which we denote as *Model A* and *Model B*.

a) Solution generation (Model A): For each language–instruction pair, *Model A* is prompted to act as a coding assistant in the target language. Its objective is to implement the specified I/O behavior as a complete program. The generated program is required to:

- read all inputs from standard input in the format implied by the instruction;
- write only the required outputs to standard output using a fixed and predictable format; and
- avoid interactive prompts, diagnostic messages, and non-deterministic behavior.

If successfully generated, the program is recorded as *canonical solution* of the task. The corresponding instruction is stored in both `instruction` and `prompt` fields of the task object.

b) Test generation (Model B): Given the instruction, *Model B* is prompted to act as a QA engineer for the same target language and to generate a list of input–output examples in JSON format. Crucially, *Model B* has access only to the natural language instruction and does not see the canonical solution. This separation ensures that test generation remains implementation-agnostic and conceptually consistent across languages. The prompt enforces the following requirements:

- a minimum number of diverse test cases;
- a single, consistent layout for all `"input"` fields, for example, "n on the first line, followed by n integers on the second line";
- `"output"` fields that match what a correct program would print for the corresponding input, without additional text.

Because LLM outputs may include extraneous commentary or formatting artifacts such as code fences, we apply a robust JSON extraction procedure to recover the test cases. If the extracted list is empty or below the required size, we re-prompt the model up to a fixed number of attempts.

c) Pre-deduplication and ingest validation: Each candidate is assigned a language-specific fingerprint $\mathrm{fp}(L, I)$ computed by normalizing the instruction text I and hashing together with the language L. If the fingerprint already exists in any split (incoming/benchmark/unverified), the candidate is discarded to avoid duplicates. Otherwise, we assign a stable per-language identifier `L/XXXX` (e.g., `Ada/0101`) and append the candidate to the *incoming* pool.

E. Program Verification and Benchmark Construction

The incoming pool contains noisy candidates, including tasks with syntax errors, underspecified I/O, or inconsistent tests. We therefore apply execution-based verification in a unified sandbox.

a) Execution environment: We build a Docker image with the necessary compilers/interpreters for all target languages and define per-language compile/run commands. For compiled languages, we compile first and record failures as `COMPILE_ERROR`. Each test case is then executed with a timeout and stdout is compared to the expected output after whitespace normalization. This yields a deterministic, reproducible acceptance test for every task.

b) Pass-k verification with feedback: We verify each candidate by attempting to solve it under the same harness. Let $\mathcal{T}$ be the test suite and $S_1, \ldots, S_k$ be up to k solution attempts produced by a verifier model. A task is **verified** iff

$$\exists i \in \{1, \ldots, k\} : \ \mathrm{Exec}(S_i, \mathcal{T}) = \texttt{OK}.$$

In practice, we optionally run the candidate's current `canonical_solution` first; if it already passes all tests, the task is immediately verified. Otherwise, we query the verifier model for up to k attempts using **full feedback** between attempts:

- If the previous attempt fails to compile, we provide a short compiler error excerpt.
- If it compiles but fails, we provide the *input* of the first failing test case (without revealing the expected output), encouraging the model to fix parsing or corner cases while avoiding label leakage.

Verification stops early once any attempt passes all tests. For every task we store verification metadata including: verifier model name, k, the number of attempts used, and timestamps.

c) Model-cascading across rounds: A single verifier may fail on solvable tasks due to idiosyncratic weaknesses. To reduce spurious rejections, we run verification in multiple rounds with different verifier models. After each round, only tasks still unverified are passed to the next verifier. A task enters the benchmark split once *any* verifier succeeds within pass-k; tasks that fail all verifiers across rounds are retained in the *unverified* pool. This cascading design preserves a clear admission criterion while improving recall of valid tasks.

F. Benchmark Statistics

We generate an incoming pool of 120 candidate tasks per language. After multi-round verification, 1,196 tasks across 12 languages are retained in the benchmark split, while 244 tasks are assigned to the unverified split, yielding 1,440 total candidates. Each verified task is paired with an I/O test suite; on average, a task contains 13.62 tests (Table I).

TABLE I
DATASET STATISTICS BY LANGUAGE. "VERIFIED" TASKS FORM THE BENCHMARK; "UNVERIFIED" ARE EXCLUDED.

Language	Verified	Unverified	Avg. #Tests	Test Range
Ada	109	11	13.80	12–24
COBOL	85	35	13.42	10–20
D	109	11	13.63	12–18
Fortran	96	24	13.36	10–20
Julia	100	20	13.87	12–20
Lisp	100	20	13.89	12–28
Lua	106	14	13.85	12–30
OCaml	105	15	13.67	12–20
Octave	97	23	12.99	10–19
Prolog	87	33	13.37	10–18
REXX	100	20	13.73	12–18
Tcl	102	18	13.77	12–24
All	**1196**	**244**	**13.62**	**10–30**

Figure 3 highlights that our benchmark only partially overlaps with prior multilingual suites, and that several of our target PLs are entirely missing from widely used evaluation stacks. Even for overlapping PLs, our benchmark provides orthogonal signals: MultiPL-E is primarily translation-based from Python-centric benchmarks, while McEval emphasizes human-authored multilingual tasks. In contrast, our construction targets legacy/long-tail ecosystems with a different execution and task-design perspective.

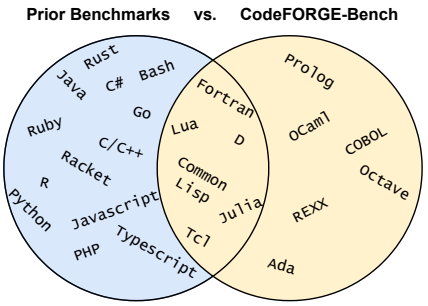

Fig. 3. Programming-language (PL) coverage of our benchmark versus prior multilingual code-generation suites (HumanEval [2], MultiPL-E [11], and McEval [12]). The overlap region contains PLs shared with at least one prior suite; we additionally introduce legacy/long-tail PLs that are absent from all three (Ada, COBOL, Octave, OCaml, Prolog, and REXX).

IV. EXPERIMENT

A. Evaluation Protocol

We evaluate `CodeFORGE-Bench` with execution-based correctness using *pass@k*. For each task, we sample 5 candidate programs from the target model (temperature 0.2) and evaluate them independently in a Dockerized runtime: compile (when applicable), then execute against the task test suite. We compute pass@k ($k \in \{1, 3, 5\}$) by marking a task as solved if any of the first k candidates passes all tests. Unless stated otherwise, the runner uses a 120s request timeout (up to 5 retries), a 30s compilation timeout, and a 10s per-test runtime limit.

B. Aggregation

Let T_ℓ be the task set for language ℓ, and let $s_{t,m}^{(k)} \in \{0, 1\}$ indicate whether model m solves task t within the first k attempts ($k \in \{1, 3, 5\}$). We report per-language pass@k:

$$\mathrm{Pass@}k(m, \ell) = \frac{1}{|T_\ell|} \sum_{t \in T_\ell} s_{t,m}^{(k)}. \tag{1}$$

To avoid languages with more tasks dominating the summary, we also report a macro-average over languages:

$$\mathrm{Overall}_{\mathrm{macro}}^{(k)}(m) = \frac{1}{|\mathcal{L}|} \sum_{\ell \in \mathcal{L}} \mathrm{Pass@}k(m, \ell), \tag{2}$$

where $\mathcal{L}$ is the set of benchmark languages.

C. Difficulty Bucketing

To study performance as a function of difficulty, we assign each task a *post hoc* difficulty label from cross-model outcomes under pass@1 protocol. Let $\mathcal{R}$ be a fixed reference set of models. For each task t, we compute the solve rate

$$r_t = \frac{1}{|\mathcal{R}|} \sum_{m \in \mathcal{R}} s_{t,m}^{(1)}, \tag{3}$$

and define a difficulty score $d_t = 1 - r_t$ (higher is harder). As languages have different base success rates, we bucket tasks *within each language* ℓ by splitting $\{d_t\}_{t \in T_\ell}$ into terciles, yielding `easy`/`medium`/`hard`. We store the bucket results in each task's metadata (field `difficulty`) for analysis.

TABLE II
BASELINE PERFORMANCE (%) OF CODEFORGE-BENCH ON 7 LLMS WITH PASS@1/@3/@5. **MACRO** = MACRO-AVERAGE.

	Ada			COBOL			D			Fortran			Julia			Lisp			Lua			OCaml			Octave			Prolog			REXX			Tcl			Macro		
	@1	@3	@5	@1	@3	@5	@1	@3	@5	@1	@3	@5	@1	@3	@5	@1	@3	@5	@1	@3	@5	@1	@3	@5	@1	@3	@5	@1	@3	@5	@1	@3	@5	@1	@3	@5	@1	@3	@5
Gem3	78.9	89.9	91.7	63.5	81.2	83.5	86.2	89.0	89.9	66.7	70.8	74.0	91.0	95.0	97.0	91.0	97.0	97.0	80.2	89.6	90.6	90.5	92.4	93.3	77.3	87.6	89.7	79.3	93.1	93.1	82.0	94.0	97.0	82.4	85.3	85.3	**80.7**	**88.7**	**90.2**
DS3.2	57.8	75.2	80.7	14.1	30.6	42.4	52.3	82.6	88.1	57.3	72.9	75.0	84.0	91.0	92.0	80.0	91.0	95.0	86.8	90.6	91.5	71.4	87.6	90.5	73.2	85.6	89.7	55.2	85.1	87.4	69.0	84.0	92.0	79.4	87.3	88.2	**65.0**	**80.3**	**84.4**
GPT5.1-Cdx	73.4	87.2	90.8	10.6	28.2	34.1	28.4	47.7	55.0	47.9	76.0	81.2	43.0	64.0	72.0	86.0	93.0	95.0	87.7	93.4	95.3	85.7	95.2	97.1	80.4	85.6	89.7	66.7	93.1	95.4	76.0	91.0	97.0	86.3	91.2	91.2	**64.3**	**78.8**	**82.8**
Grok4	53.2	77.1	78.0	27.1	36.5	41.2	69.7	88.1	89.9	61.5	69.8	76.0	47.0	64.0	66.0	83.0	92.0	93.0	73.6	85.8	87.7	73.3	88.6	88.6	69.1	81.4	84.5	41.4	64.4	69.0	48.0	82.0	87.0	77.5	84.3	87.3	**60.4**	**76.2**	**79.0**
ClS4.5	60.6	67.9	67.9	14.1	17.6	21.2	56.9	58.7	59.6	54.2	58.3	59.4	61.0	63.0	63.0	83.0	85.0	86.0	75.5	79.2	79.2	60.0	66.7	67.6	45.4	47.4	47.4	39.1	43.7	50.6	53.0	63.0	65.0	76.5	80.4	80.4	**56.6**	**60.9**	**62.3**
GLM4.6	26.6	42.2	48.6	9.4	15.3	20.0	16.5	33.9	44.0	30.2	42.7	59.4	64.0	79.0	83.0	46.0	65.0	76.0	58.5	77.4	82.1	33.3	49.5	61.0	25.8	36.1	47.4	12.6	21.8	32.2	35.0	53.0	62.0	66.7	77.5	81.4	**33.7**	**49.5**	**58.1**
Qwen3	33.0	43.1	45.0	1.2	9.4	15.3	14.7	22.0	25.7	54.2	59.4	61.5	36.0	40.0	40.0	46.0	63.0	69.0	58.5	65.1	66.0	46.7	56.2	57.1	46.4	52.6	53.6	30.1	43.7	52.9	47.0	61.0	66.0	55.9	65.7	70.6	**39.1**	**48.4**	**51.9**

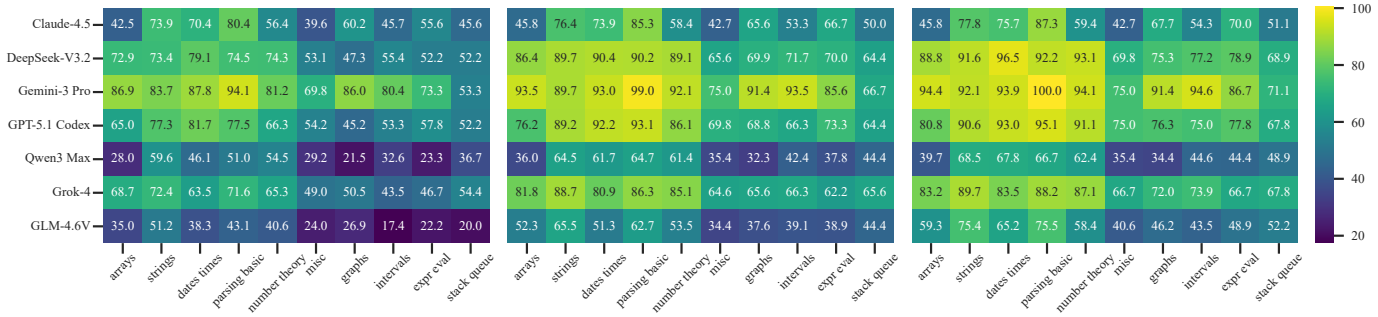

Fig. 4. Topic-wise performance (%) across 10 task topics for the evaluated LLMs. **Left:** pass@1. **Middle:** pass@3. **Right:** pass@5.

D. Setup and reported scope

We evaluate seven LLMs via OpenRouter: anthropic/claude-sonnet-4.5, deepseek/deepseek-v3.2-speciale, google/gemini-3-pro-preview, openai/gpt-5.1-codex-max, qwen/qwen3-max, x-ai/grok-4, and z-ai/glm-4.6v. We report results on 12 benchmark languages (Ada, COBOL, D, Fortran, Julia, Lisp, Lua, OCaml, Octave, Prolog, REXX, Tcl) using the Dockerized pass@5 harness described above.

V. RESULTS

A. Overall performance on 12 rare/legacy languages

Table II summarizes macro-averaged execution-based performance across the 12-language reported subset (each language weighted equally). Overall, `Gemini 3 Pro` is the strongest model, achieving the best macro pass@1/3/5 (80.7%/88.7%/90.2%). `DeepSeek v3.2 Speciale` and `GPT 5.1 Codex-Max` form the second tier with comparable pass@5 (∼83–84%), while `Claude Sonnet 4.5` and `Grok 4` occupy the third tier (∼62–79%). `Qwen3-Max` and `GLM 4.6v` lag behind substantially, with pass@1 below 40%.

A notable trend is the *sample-efficiency gap*: some models obtain large gains when increasing the number of attempts from 1 to 5. For example, `GLM 4.6v` improves by over 24 percentage points (33.7%→58.1%), whereas `Gemini 3 Pro` gains only 9.5 points. This suggests that weaker models may still contain correct solutions in their candidate set but exhibit higher variance across stochastic generations, making them potentially more sensitive to search/reranking strategies.

We also observe a *performance ceiling*: even the best model saturates below 91% pass@5, compared to near-perfect scores often reported on Python benchmarks such as HumanEval. This gap underscores that rare/legacy languages pose fundamentally harder challenges, whether due to sparser training data, stricter syntax constraints, or less forgiving toolchains.

Key findings.

- **Strong overall leader:** `Gemini 3 Pro` consistently ranks first across pass@1/3/5, with a 15+ point margin over second-tier models at pass@1.
- **High-variance models:** `GLM 4.6v`, `Grok 4`, and `GPT 5.1 Codex-Max` benefit disproportionately from multiple attempts, indicating larger per-attempt variance and potential gains from better decoding strategies.
- **Performance ceiling:** No model exceeds 91% pass@5, revealing persistent challenges in rare/legacy settings that high-resource benchmarks do not capture.
- **Language-specific brittleness:** Performance gaps across languages are substantial (up to 40+ points within a single model), implying that rare/legacy settings expose weaknesses invisible in Python-centric evaluations.

B. Cross-language variation and robustness

Table II reports per-language pass@1/3/5 over the 12-language subset. We observe substantial cross-language variation shared across models, with per-language pass@1 spanning from 1.2% (`Qwen3-Max` on COBOL) to 91.0% (`Gemini 3 Pro` on Julia/Lisp), a nearly 90-point range that underscores the heterogeneity of rare/legacy language difficulty.

COBOL is consistently the hardest language. In the pass@1 breakdown, most models drop sharply on COBOL compared to other languages: five models score below 30% pass@1, with `Qwen3-Max` achieving only 1.2%. Even the strongest model (`Gemini 3 Pro`) reaches just 63.5%

pass@1: over 25 points below its average on other languages. This aligns with the intuition that COBOL tasks are both syntactically rigid and toolchain-sensitive (compile/link/runtime quirks), amplifying failure rates. The fixed-column formatting, verbose division structure, and strict data type declarations create compounding error sources that LLMs struggle to satisfy simultaneously.

Easiest languages cluster around dynamic and functional ecosystems. Lisp, Lua, Tcl, and Julia consistently show higher pass rates across models, with top performers exceeding 90% pass@5. These languages share characteristics that favor LLM generation: flexible syntax, forgiving runtimes, and lightweight I/O conventions. Notably, OCaml, despite being statically typed, also performs well, suggesting strong type inference and concise syntax may partially offset compilation strictness.

Compiled vs. interpreted gap. A secondary pattern emerges when grouping by execution model: interpreted languages (Lua, Tcl, Lisp, Octave) average 7–12 points higher pass@1 than compiled languages (Ada, COBOL, D, Fortran) across most models. This gap reflects not only syntactic differences but also the absence of a separate compilation phase where errors can accumulate before execution.

No single model dominates every language. While `Gemini 3 Pro` is the best overall performer, `GPT 5.1 Codex-Max` attains the best pass@1 on Lua (87.7%), Tcl (86.3%), and Octave (80.4%), and `DeepSeek v3.2 Speciale` leads on Lua at pass@5 (91.5%). This highlights that cross-language robustness is not perfectly correlated with overall ranking. Models exhibit complementary strengths, suggesting potential gains from ensemble or routing strategies.

Key findings.

- **Extreme variance:** Per-language pass@1 spans nearly 90 points (1.2%–91.0%), revealing that "rare/legacy" is not a monolithic category but a spectrum of difficulty.
- **COBOL as stress test:** COBOL uniquely stresses LLM code generation, with most models failing on >70% of tasks at pass@1, making it a valuable diagnostic for toolchain and syntax robustness.
- **Interpreted advantage:** Dynamic/interpreted languages consistently outperform compiled counterparts, suggesting that compilation strictness amplifies LLM error rates.
- **Complementary model strengths:** No model dominates all languages; cross-model routing could exploit language-specific expertise.

C. Topic-level difficulty patterns

We report a coarse topic-level breakdown by aggregating tasks into 10 task topics. Figure 4 shows that data-structure-centric and execution-sensitive topics (e.g., stack/queue, I/O, interval logic, expression evaluation, graphs) are systematically harder than text-centric tasks (string manipulation and basic parsing). A likely driver is that these topics demand sustained state tracking, invariant maintenance, boundary-case reasoning, and exact control over execution order and output formatting, where small deviations compound into failures. In contrast, many text-centric tasks reduce to more local transformations with stronger surface cues and fewer interacting constraints. This pattern supports the hypothesis that rare/legacy settings magnify brittleness in stateful reasoning and strict I/O adherence.

D. Failure mode analysis

Beyond pass@k, we analyze *how* models fail by aggregating per-attempt outcomes into coarse categories (OK, WRONG_ANSWER, COMPILE_ERROR, RUNTIME_ERROR). Figure 5 reveals qualitatively different failure profiles, which are informative because each error type maps to a distinct bottleneck in rare/legacy code generation.

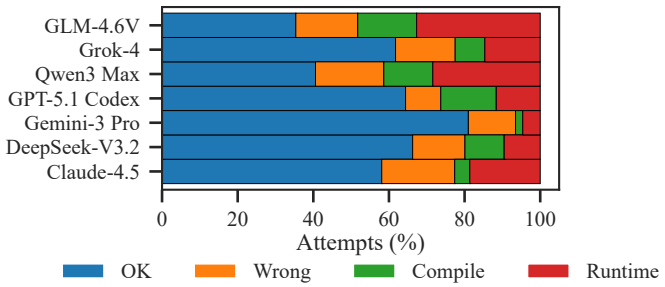

Fig. 5. Outcome distribution per model aggregated over the 12 reported languages and all attempts.

Compile-heavy failures suggest syntax/toolchain mismatch. Some models exhibit more compilation failures, consistent with difficulties in producing language-idiomatic boilerplate, correct module/import usage, and toolchain-compatible build artifacts under strict harnesses. In legacy languages, minor deviations (e.g., missing pragmas, incorrect file structure, nonstandard compiler flags) can be unrecoverable, making compilation rate a proxy for "environment alignment."

Runtime-heavy failures indicate semantic fragility. Models with fewer compile errors may still fail frequently at runtime due to incorrect I/O handling, missing corner cases, or subtle language-specific semantics (e.g., integer division, string parsing, or backtracking behavior in logic languages). These failures often arise from stateful control flow and edge cases that are easy to under-specify in single-shot generation.

Wrong-answer failures reflect reasoning gaps under legacy constraints. Even when code runs, correctness can drop due to algorithmic errors or mismatch between the generated solution and the exact input/output contract which is an issue exacerbated when the language's standard library differs from mainstream defaults. Notably, wrong-answer dominates when models can "speak" the syntax but struggle to satisfy stricter problem constraints end-to-end.

E. Scaling with task difficulty

To test whether rare and legacy languages amplify hardness effects, Figure 6 reports pass@1/3/5 stratified by difficulty (easy, medium, hard). All models degrade as difficulty increases, but the rate of degradation differs.

Robust models degrade more gracefully. Top-tier models maintain comparatively strong pass@1/3/5 on hard tasks, suggesting better algorithmic generalization and stronger control over language-specific details even under stricter constraints.

Weaker models collapse on hard tasks. Weaker models show a pronounced drop from easy to hard, consistent with compounding failures that combine incorrect reasoning with brittle realization in legacy languages and I/O contracts.

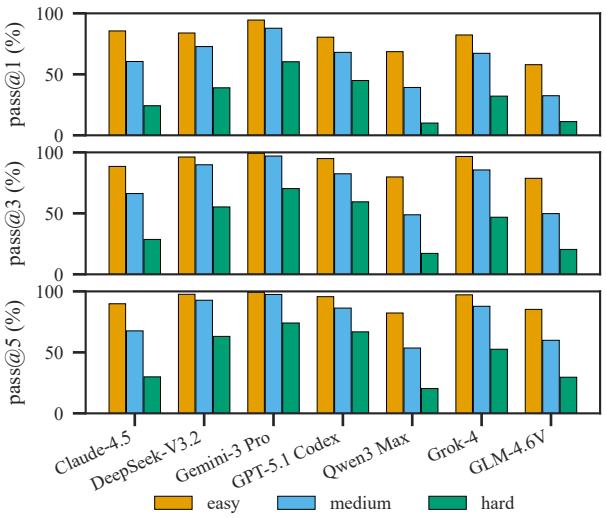

Fig. 6. Pass@{1,3,5} (%) stratified by difficulty (easy/medium/hard), aggregated over the 12 reported languages (k=5 attempts per task). Top/middle/bottom: pass@1, pass@3, pass@5.

The three panels separate *first-shot robustness* (pass@1) from *multi-sample recoverability* (pass@3/5). When moving from pass@1 to pass@5, improvements indicate that a model can self-correct through sampling even under strict legacy-language constraints, while the remaining gap on `hard` tasks highlights persistent brittleness beyond simple resampling.

VI. Limitations

`CodeFORGE` is currently limited to command-line, single-file programming tasks that assume standard I/O conventions. It does not cover languages and ecosystems such as MATLAB and ABAP, or other domain-specific toolchains that rely on proprietary runtimes, specialized frameworks, graphical interfaces, or heavyweight build systems. Moreover, while the multi-LLM cascading strategy substantially improves benchmark reliability, it also increases token consumption and operational cost, making it necessary to balance iterative feedback depth with overall computational efficiency when scaling to larger and more heterogeneous task sets. Future work will extend `CodeFORGE` to support multi-module project structures and external library dependencies, thereby relaxing the current single-file constraint and broadening its applicability.

VII. Conclusion

This paper presents `CodeFORGE`, an automated benchmark synthesis framework that narrows the evaluation gap for low-resource and legacy programming languages. To address the scarcity of high-quality datasets for these languages, we propose a multi-LLM cascading verification strategy combined with iterative feedback and cross-validation in isolated Docker environments, enabling the generation of reliable and functionally correct programming tasks. We release `CodeFORGE-Bench`, covering 12 languages, as a standardized benchmark for assessing the generalization performance of modern LLMs on diverse programming ecosystems.

Acknowledgment

This work was supported in part by NYU Shanghai Serendipity Fellowship, Google Ph.D. Fellowship and NYUAD Center for Cyber Security (CCS), funded by Tamkeen under the NYUAD Research Institute Award G1104.

References

[1] M. Shao *et al.*, "Survey of different large language model architectures: Trends, benchmarks, and challenges," *IEEE Access*, vol. 12, pp. 188 664–188 706, 2024.

[2] M. Chen *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[3] J. Austin *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[4] D. Hendrycks *et al.*, "Measuring coding challenge competence with APPS," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.

[5] Y. Li *et al.*, "Competition-level code generation with AlphaCode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[6] J. Liu *et al.*, "Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[7] L. Yang *et al.*, "On the evaluation of large language models in unit test generation," *arXiv preprint arXiv:2406.18181*, 2024.

[8] S. Ravi and M. Coblenz, "An empirical evaluation of property-based testing in python," *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA2, pp. 3897–3923, 2025.

[9] I. Magar and R. Schwartz, "Data contamination: From memorization to exploitation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 2022, pp. 157–165.

[10] Q. Zheng *et al.*, "Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023.

[11] F. Cassano *et al.*, "Multipl-e: A scalable and polyglot approach to benchmarking neural code generation," *IEEE Transactions on Software Engineering*, vol. 49, no. 7, pp. 3675–3691, 2023.

[12] L. Chai *et al.*, "Mceval: Massively multilingual code evaluation," in *International Conference on Learning Representations (ICLR)*, 2025.

[13] M. A. M. Khan *et al.*, "xcodeeval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation, and retrieval," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024.

[14] S. Lu *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.

[15] R. Baltaji *et al.*, "Cross-lingual transfer in programming languages: An extensive empirical study," *Transactions on Machine Learning Research*, 2025.

[16] Y. Wang *et al.*, "Self-instruct: Aligning language models with self-generated instructions," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL)*, 2023.

[17] C. Xu *et al.*, "Wizardlm: Empowering large pre-trained language models to follow complex instructions," in *International Conference on Learning Representations (ICLR)*, 2024.

[18] A. Singh *et al.*, "Beyond human data: Scaling self-training for problem-solving with language models," *Transactions on Machine Learning Research*, 2024.

[19] H.-S. Li, P. Fernandes, I. Gurevych, and A. F. T. Martins, "Doce: Finding the sweet spot for execution-based code generation," *arXiv preprint arXiv:2408.13745*, 2024.

[20] X. Chen *et al.*, "Teaching large language models to self-debug," in *International Conference on Learning Representations (ICLR)*, 2024.