Terminal-Bench-LILT: Multilingual Agentic Coding Benchmark Grounded in Language, Region, and Culture

Yunsu Kim **Kaden Uhlig** **Ashwin Purohit** **Milind Agarwal** **Patrick Simianer**
Anil Arslan **Kiarash Mokhtari** **Thomas Zenkel** **Johannes Mosig**
Gabriel Bretschner **Shamik Bose** **Joern Wuebker** **John DeNero**

LILT, Inc.
contact@lilt.com

Abstract

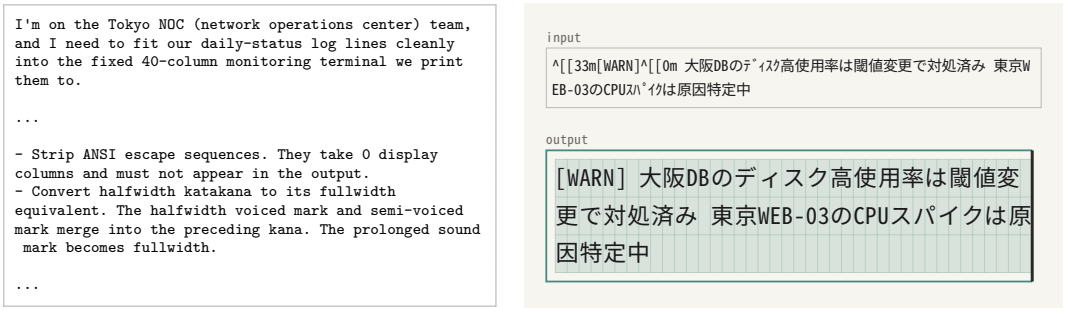
Most evaluations for coding agents are conducted exclusively in English, which does not reflect real-world multilingual deployment. We present Terminal-Bench-LILT, a suite of 300 authentic coding tasks in 10 languages: Arabic, Czech, German, Spanish, Hindi, Japanese, Korean, Serbian, Turkish, and Chinese. Each task targets issues specific to non-English software development that have no direct English equivalent, e.g., internationalization, encoding, text normalization, and cultural conventions. All tasks are authored by native-speaker programmers and validated through a multi-stage quality control pipeline. Evaluation of six frontier models reveals that even the strongest model reaches only 63.1% pass rate, with many tasks unsolved by any model. Performance varies substantially by language and does not track general coding benchmark rankings, highlighting that multilingual coding competence is a distinct and underexplored capability axis. Sample tasks are available at `https://github.com/lilt/terminal-bench-lilt`.

1 Introduction

Software development is a global field in which English serves as the common working language: open-source projects are coordinated in English (Kocetkov et al., 2023; Lozhkov et al., 2024; Bhuiyan et al., 2026), and community resources such as Stack Overflow are written and answered largely in English (Stack Overflow, 2024). Most programmers therefore read, write, and discuss code in English, and the models and tools built on this ecosystem are trained and tuned primarily for English and perform best in it (GitHub, 2025b). The benchmarks used to evaluate these agents follow the same pattern: beyond being written almost entirely in English, they implicitly encode the data, locales, and conventions of an English-speaking development context (Chen et al., 2021; Jimenez et al., 2024; Jain et al., 2025; Merrill et al., 2026).

However, many developers are not native English speakers (SlashData, 2025; GitHub, 2025a) and solve problems outside an English-speaking context. Particularly in local markets, developers work with native-language data, configure locale-specific settings, and build services that fit local conventions and culture. These are real engineering challenges that rarely surface in English-speaking contexts, and they directly shape how non-English-speaking developers experience coding agents. The ideal coding agent proactively identifies and resolves implicit technical and cultural assumptions. This is taken for granted in English-centric tasks, while in non-English contexts agents need explicit guidance to avoid pitfalls native speakers consider obvious (Shen et al., 2024; Naous et al., 2024; Myung et al., 2024). This raises the expertise required of users and makes the agent tedious to use.

No existing coding benchmark evaluates agents on this dimension, so we built Terminal-Bench-LILT, a suite of 300 terminal-based coding tasks in 10 languages across three difficulty tiers. Native programmers authored each task from a problem they had actually encountered, then put it through layered quality control combining automated checks and human review.



(a) Instruction

(b) Example input/output.

Figure 1: The `ja-terminal-width` task from Terminal-Bench-LILT. The instruction in (a) is abridged for clarity. In (b), each grid cell is one display column of the terminal.

Figure 1 shows a task from Terminal-Bench-LILT. The agent is asked to fit Japanese log lines into a fixed 40-column terminal. Each line mixes Japanese characters, ASCII, emoji, and terminal color escapes, which occupy different numbers of columns. Further edge cases arise naturally from Japanese text itself, for example:

1. Unicode variation selectors, often used in names to force a specific visual variant, are not displayed and must be left out of the width calculation. Counting code points, as Python's `len()` does, gives each one a column.
2. Kanji have compatibility ideographs that look identical but have different bytes, e.g., 神 = U+FA19 and 神 = U+795E. Applying Unicode NFKC without care merges them into the more common form, altering text the task only asked to reformat.

Such issues are routine for a developer who handles Japanese terminal output and tedious to spell out. The instruction therefore asks for it only implicitly, while the tests check exactly these points. The task thus precisely measures a smooth user experience for a native developer working in Japanese.

The rest of the paper is organized as follows. Section 2 reviews prior coding benchmarks and related work. Section 3 describes the design, construction, and quality control of Terminal-Bench-LILT. Section 4 evaluates frontier models on the benchmark and analyzes the results. Section 5 concludes, followed by an appendix listing sample tasks.

2 Related Work

Agentic Coding Benchmarks Evaluation of code generation has moved from single-function synthesis (Chen et al., 2021) toward repository-level and agentic settings: SWE-bench (Jimenez et al., 2024) poses real GitHub issues, LiveCodeBench (Jain et al., 2025) provides contamination-free competitive problems, and Terminal-Bench (Merrill et al., 2026) evaluates agents on command-line tasks. These benchmarks are all English-only; we build on Terminal-Bench and extend its task format to ten non-English languages.

Multilingual Coding Benchmarks Many coding benchmarks that call themselves *multilingual* vary the *programming* language of the code while keeping the instruction in English (Cassano et al., 2023; Athiwaratkun et al., 2023; Zan et al., 2025). Others do vary the human language, but only in the instruction, translating English coding tasks into other natural languages (Raihan et al., 2025; Peng et al., 2024; Wang et al., 2023; 2024; Hofman et al., 2025). They primarily measure model comprehension of translated prompts, not how useful a model is for developers in a non-English context.

In coding, this comprehension axis seems to be an inconsistent driver of difficulty. For instance, Hofman et al. (2025) translate SWE-bench (Jimenez et al., 2024) into ten languages, but the mean non-English performance shows only a 1.3% relative drop from English. Meanwhile, Wang et al. (2024) report a relative decrease of at least 13% in pass rate

on HumanEval-X (Zheng et al., 2023) tasks translated into Chinese. In these cases, it is hard to distinguish between errors introduced by translation and genuine limits of model capability; moreover, all tasks are drawn from an English context.

Our benchmark is orthogonal to both lines of work: the programming language is incidental, and we vary the human language together with its regional and cultural context. Our tasks are authored directly in a given language by native speakers, avoiding translation artifacts, and we keep only tasks that are genuinely language- and culture-specific, enabling a targeted evaluation of models' usefulness for non-English developers.

Reasoning Language One hypothesis is that a translated instruction leaves the underlying problem unchanged: a model can read a non-English prompt yet still reason in English (Schut et al., 2025). Barua et al. (2026) find that reasoning in English outperforms reasoning in the target language, with the gap widening on multi-step tasks. For code generation, Nishigata et al. (2025) exploit this directly, tuning models to attach an English chain-of-thought to non-English instructions. Difficulty from the instruction language is thus limited, and such tuning can mitigate it further. Our tasks instead rest on the locale-specific knowledge the instructions leave implicit — a gap that reasoning alone does not close (Section 4.3).

Cultural and Locale Knowledge Benchmarks of everyday cultural knowledge expose large disparities between well- and under-represented cultures: Myung et al. (2024) report a best-to-worst culture gap of up to 57 percentage points for GPT-4 on short-answer questions, and related work documents cultural bias (Naous et al., 2024) and the limits of cultural commonsense (Shen et al., 2024) in LLMs. These benchmarks, however, probe such knowledge in plain question answering, not in the work that models are actually used for. Our benchmark tests the same knowledge in agentic coding, where an agent passes only if its code handles the local convention correctly.

3 Terminal-Bench-LILT

Terminal-Bench-LILT has 300 tasks spanning 10 languages, with 30 tasks in each language. Each task runs under Harbor (Harbor Framework Team, 2026) in an isolated environment with deterministic verification. It follows the original Terminal-Bench task structure (Merrill et al., 2026), with additional fields and documentation for multilingual challenges:

- **Instruction** describes a real-world coding problem requiring the agent to write a solution script. It is provided in both the native language and English.
- **Environment** (Docker configuration) together with any native-language data assets required by the task.
- **Verifier** with deterministic tests.
- **Oracle solution** accepted by the verifier.
- **Metadata** containing brief descriptions of the task, solution, and verifier, together with the ISO 639-1 language code, challenge category and subcategory, difficulty, and infrastructure configuration.
- **README** explaining the motivation behind the task design, its realism, and its linguistic and technical challenges in detail.

Instructions were written *from scratch in the native target language* first. Many tasks are grounded in locale-specific data, such as files in legacy encodings and regional date formats, and are therefore most naturally described in the language in which the data was produced. This approach also reflects how users instruct coding agents in their native languages, using colloquial phrasing and domain-specific vocabulary. Writing the instructions in English first and then translating them, as is common in prior work (Section 2), could introduce translation artifacts and make the resulting tasks less representative of real-world use.

An English translation of the native instruction is also provided, making it accessible to non-native speakers and enabling controlled experiments that isolate the effect of instruction language from task difficulty (Section 4.3).

Category	Subcategory	Scope / Examples
Internationalization	Locale-based Templating	Plurals, numbers, suffixes, date formats, Korean *man* numbering
	Character Rendering	RTL, Arabic shaping, Simplified vs. Traditional Chinese, Cyrillic
Interaction with Environment	Encodings	Non-Unicode/legacy file encodings, conversion
	System Configuration	CJK input method debugging, RTL shell tools, locale-correct sorting
Text Conversion	Normalization	NFC/NFKC normalization, mojibake correction
	ML & Data Processing	Case augmentation, tokenizer training, typo simulation
	Variant Conversion	Script conversion, German spelling reform
Text Handling	Document Search	Stemming, stop word removal for agglutinative languages
	Unicode Confusion Lists	Anti-spam character confusion in Cyrillic-first applications
	User Authentication	Normalized username/password comparison, DB migration
Cultural & Other	Calendar/Date Conversion	Lunar, Hijri, Umm al-Qura, holiday calculation
	Other	Culture/Region-specific issues that cannot be categorized into existing subcategories

Table 1: Challenge taxonomy for Terminal-Bench-LILT tasks.

Category	AR	CS	DE	ES	HI	JA	KO	SR	TR	ZH	Total
Internationalization	10	7	3	2	7	5	6	6	5	1	52
Interaction with Environment	2	2	5	3	3	3	1	1	2	3	25
Text Conversion	10	3	3	11	6	7	2	15	12	7	76
Text Handling	1	1	2	2	4	2	1	3	2	0	18
Cultural & Other	7	17	17	12	10	13	20	5	9	19	129
Total	30	30	30	30	30	30	30	30	30	30	**300**

Table 2: Distribution of tasks by challenge category and language.

3.1 Task Design

Each task includes one or more challenges specific to its target language, region, or culture. We assign every task a primary category and subcategory from the challenge taxonomy in Table 1; Table 2 shows the distribution of tasks by category and language. Descriptions of sample tasks are given in Appendix A.

Regardless of category, every task follows three design principles:

1. A task should reflect a problem that a programmer in the target locale could plausibly encounter. Its data, formats, and workflows should match those *actually used in practice*.
2. A task's difficulty should arise from *locale-specific elements*, not from general engineering complexity. A task that merely restates an ordinary programming problem in another language does not qualify.
3. Locale-specific challenges should be *embedded implicitly* in the engineering problem rather than stated directly. This tests whether agents can supply the local knowledge that native developers take for granted, without requiring users to spell it out.

```
I have a small corpus of Czech sentences at /app/data/
sentences.jsonl.

I need a script at /app/split.py that reads /app/data/
sentences.jsonl and writes /app/output/lines.jsonl with
one record per input sentence, breaking each sentence into
 one or more typeset lines that fit a configured target
width.

...

The lines must follow Czech orthotypography - the
canonical rules a competent Czech typographer would apply.
 The description below is illustrative only; there are
further patterns too, and all of them apply.

Single-letter prepositions and conjunctions (`k`, `s`, `v
`, `z`, ...) must never end a line; wherever the original
text has one as a standalone token followed by another
word, I glue them with U+00A0 (NBSP) instead of an ASCII
space. The same non-breaking space joins a numeric
quantity to its unit symbol, the section sign to a
following article number (`§ 12`), titles before a name (`
Dr.`, `Mgr.`, ...), ...

The bundled sentences sample only a handful of these
patterns; since the script will also run over a different
file, I implement the canonical Czech rules in full rather
 than overfitting to the bundled sentences.

...
```

(a) `cs-nbsp-justify`

```
The file /app/produkte.csv contains a list of products
from our wholesale catalogue (columns: `produkt_id`, `
bezeichnung`). For accounting, each product needs the
applicable German VAT rate - either the reduced rate (7%)
or the standard rate (19%).

A working student left a first draft at /app/
klassifikator_draft.py. In a spot-check, our tax
department flagged several incorrect classifications, so
the draft is not acceptable as-is.

Please write a corrected script /app/klassifikator.py that
 reads the input file and assigns each row the correct VAT
 rate under German tax law as of 1 January 2026. Write the
 result as a JSON array to /app/output/ergebnis.json. Each
 entry must have the fields `produkt_id`, `bezeichnung`,
and `mwst_satz`, where `mwst_satz` is the integer `7` or
`19`. The order must match the input file.

Example output:

[
  {"produkt_id": "P001", "bezeichnung": "Roggenbrot 1kg",
"mwst_satz": 7},
  {"produkt_id": "P002", "bezeichnung": "Rotwein trocken
0,75l", "mwst_satz": 19}
]

Run with `python3 /app/klassifikator.py`, no arguments.
The directory /app/output/ will be created if needed.
```

(b) `de-umsatzsteuer-klassifikation`

Figure 2: Instructions for two sample tasks with different challenge scopes: (a) a bounded challenge and (b) an unbounded one fenced to a finite range.

The third principle is particularly important for cultural challenges where the task involves everyday knowledge in the target region beyond technical conventions such as encoding or sorting order. When the required knowledge is too specialized, e.g., a traffic rule specific to one state, the task becomes asking whether a model happens to know it, even though modern agents can retrieve such facts through web search or reference files. We avoid such esoteric knowledge recall and focus on native user experience by keeping the two rules below:

- Any challenge should be *widely known* to ordinary adults or programmers in the target country.
- When an expert domain such as law or medicine is involved, it should serve only as realistic *context*, never as the challenge itself. This is realized by citing or attaching the exact reference in the instruction.

3.2 Test Design

Each task comes with a test suite combining basic integrity checks (e.g., whether a solution script was created), sample probes using the files provided with the task, and unseen probes using new inputs. The unseen probes test generalization and discourage hard-coded solutions, but remain within the intended challenge.

Every tested behavior must be *inferable* from either the instruction, the input data, or common knowledge in the target locale. The instruction should make clear that the provided files are examples rather than the complete input set; otherwise, unseen probes would test behavior that cannot be inferred from the instruction. Requirements unrelated to the challenge, such as the output format, must be stated unambiguously.

The scope of each challenge must also be testable. When the challenge covers a *bounded* set of cases, the instruction may show only some of them and withhold the rest for testing, requiring a general solution. For example, in Figure 2a, the Czech non-breaking-space rules form a finite, closed set, so the instruction gives only a few example patterns and a small sample of sentences; the held-out sentences exercise the patterns the samples never show, catching solutions that overfit to the visible ones. When the cases are *unbounded*, the

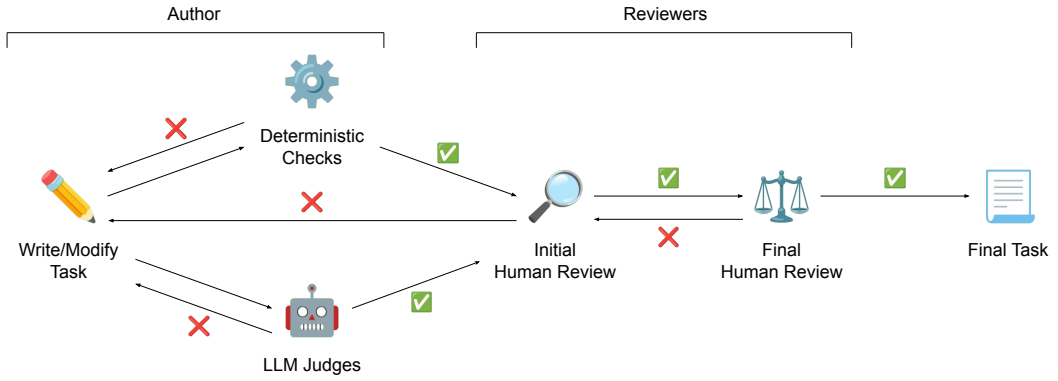


Figure 3: Quality control workflow for task submissions. ✓ and ✗ indicate whether a task passes or fails each stage, respectively. The responsible party is shown above each stage.

instruction and tests must instead define a finite range; otherwise the challenge is excluded. For example, in Figure 2b, the German VAT rates are unbounded legislated carve-outs that no general solution captures, so the instruction fixes a finite range (a specific legal snapshot and product list) and the tests check only that range.

3.3 Quality Control

Every task passes four review layers before entering the benchmark. The layers are ordered by cost: automated checks run on every commit by the task author, and human reviewers see only the tasks that survive them. Figure 3 shows the full workflow for quality control.

1. **Deterministic checks** validate the task's structure and execution setup, including its file layout, Docker environment, metadata fields, and whether the oracle solution passes all tests.
2. **LLM judges** use Gemini 3 Flash to assess the task's semantic properties, such as parity between the native-language and English instructions, alignment with the claimed task category, and whether agents can infer each tested behavior from the task materials or common local knowledge.
3. **Initial human review** re-audits the automated checks and assesses criteria that are difficult to automate: e.g., compliance with the AI-use policy, realism of the coding scenario, and clarity of constraints.
4. **Final human review** independently repeats the initial review, checks for overlap with existing tasks, and curates the final set for balanced topic and difficulty distributions, recommending difficulty adjustments when needed.

At any stage, if we found that a task could not be made valid and realistic with reasonable effort, we rejected it. Overall, about 25% of reviewed tasks were rejected.

Review Agents Reviewing a task at this level of detail requires substantial time and sustained attention. We therefore built two review agents to assist human reviewers: one assesses each task on its own, while the other compares it with existing tasks in the benchmark. Each delegates subtasks to specialized agents and combines their findings into a single report with concrete citations. We equipped the agents with deterministic tools whenever possible, e.g., to prepare execution logs and failure summaries, so the LLMs can focus on judgment. Since deployment, we have periodically used human reviewer feedback on agent outputs to refine the rubrics, aligning automated review with human judgment and reducing review time.

Tooling To streamline the multi-stage review process for hundreds of tasks, we developed an internal management application *Benchito* which tracks the stage of each submitted task

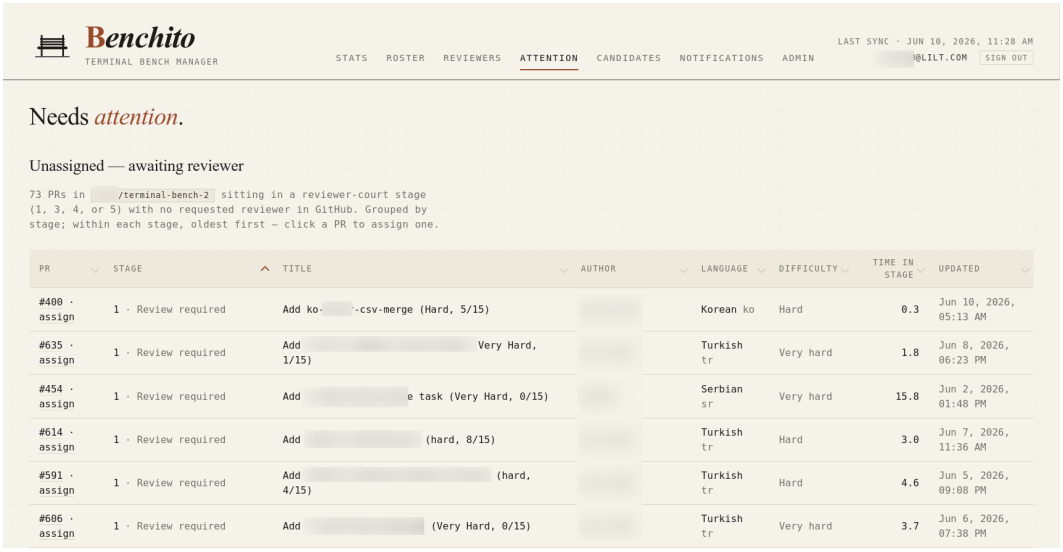


Figure 4: Workflow management tool for Terminal-Bench-LILT.

(Figure 4). Program managers can filter and sort tasks by language, difficulty, author, or reviewer and compare task counts for a selected period with the target distribution. It is integrated with the project's communication channels and notifies the responsible author or reviewer when a task changes state or fails an automated check. Access is restricted to the organization, with an audit trail of every action for accountability.

3.4 Team Management

Task quality depends on the expertise of both authors and reviewers. We therefore used explicit selection criteria and structured training.

Author Selection For each language, we screened native-speaker candidates for backgrounds in computer science or engineering and for multilingual experience. Candidates who met these criteria completed an assessment of their ability to design tasks that challenge LLMs, which our reviewers evaluated manually. Fewer than 10% of these candidates passed.

Reviewer Selection Initial reviewers were either research scientists with backgrounds in machine learning and NLP or task authors who had developed more than ten Terminal-Bench-LILT tasks. Final reviewers were senior research scientists and engineers with extensive LLM benchmarking experience.

Onboarding and Training New task authors completed a first-day checklist covering access provisioning and tool setup. Each then attended a one-hour training session on task architecture, Harbor, the taxonomy, example tasks, quality criteria, and the task submission workflow. Before authoring tasks, they were required to demonstrate their understanding of the training materials, and reviewers provided detailed feedback on their first submission. Reviewers remained available through a real-time channel for ongoing feedback and held targeted calibration sessions when needed.

3.5 Semi-Automatic Task Creation

For experimental purposes and to supplement our dataset, we created a limited number of tasks through a partially automated workflow. We built an agent equipped with specialized skills that operates in two stages:

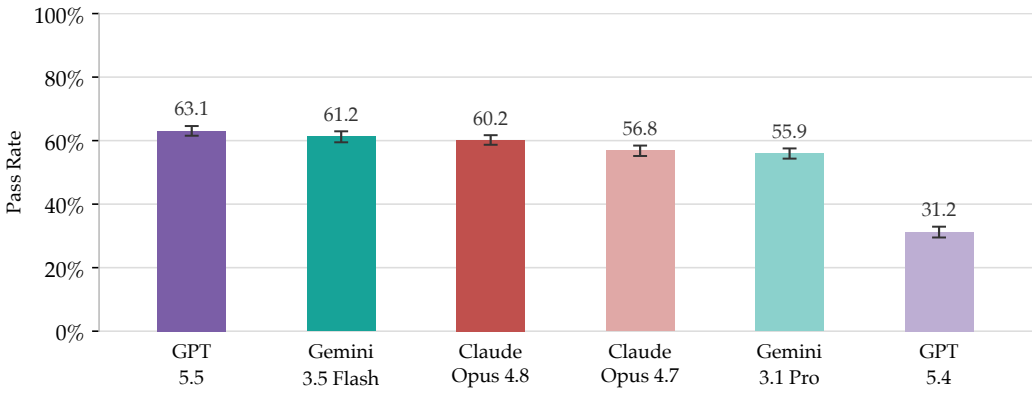


Figure 5: Task pass rate per model on Terminal-Bench-LILT, sorted by rate. Error bars show 95% confidence intervals.

1. **Idea generation.** First, the agent researches linguistic and cultural software conventions from the web, extracts candidate challenges, and removes ideas similar to accepted or rejected tasks. The remaining ideas are ranked by general quality and challenge specificity; a human reviewer approves the shortlist before development begins. This stage runs once per language.
2. **Task generation.** For each approved idea, the agent produces a progress file with a development checklist. Following this checklist iteratively, it implements the solution and unit tests, runs automated checks, and calibrates difficulty from agent trajectories.

After calibration, each created task undergoes the same automated checks and review process as tasks written by humans (Section 3.3), with an additional initial stage in which a native speaker of the target language reviews only its linguistic quality. This process produced approximately 10% of the Terminal-Bench-LILT tasks.

4 Experiments

On the full Terminal-Bench-LILT set, we evaluated six frontier LLMs across three providers: Claude Opus 4.7 (Anthropic, 2026a) and 4.8 (Anthropic, 2026b) from Anthropic, GPT-5.4 (OpenAI, 2026a) and 5.5 (OpenAI, 2026b) from OpenAI, and Gemini 3.1 Pro (Google DeepMind, 2026a) and 3.5 Flash (Google DeepMind, 2026b) from Google. Thinking/Reasoning level was set to the provider's default for each model, following the same convention as Merrill et al. (2026). The agent was fixed to the `terminus-2` harness and each task-model pair was run five times. We also manually inspected the failing trials to distinguish true language-specific failures from infrastructure issues, e.g., random crashes, OOMs, API errors, or harness failures; any trial affected by the latter was re-run so that all reported results reflect genuine task attempts.

4.1 Main Results

Figure 5 shows the aggregate pass rate across all trials for each model. GPT-5.5, Gemini 3.5 Flash, and Claude Opus 4.8 form the leading group, while Claude Opus 4.7 and Gemini 3.1 Pro follow, with GPT-5.4 far behind. The newer OpenAI model improves dramatically over its predecessor (a 31.9-point gain), while Google's and Anthropic's newer models improve more modestly, by 5.3 and 3.4 points. Even the strongest model fails 36.9% of trials, indicating that the benchmark is not saturated and leaves substantial room for improvement.

Difficulty	AR	CS	DE	ES	HI	JA	KO	SR	TR	ZH	**Total**
Easy	16	17	13	12	8	13	12	16	17	9	133
Medium	9	4	6	12	12	8	9	7	6	13	86
Hard	5	9	11	6	10	9	9	7	7	8	81
Total	30	30	30	30	30	30	30	30	30	30	**300**

Table 3: Distribution of tasks by language and empirical difficulty.

Model	AR	CS	DE	ES	HI	JA	KO	SR	TR	ZH	σ_{lang}
GPT-5.5	69.3	60.7	51.3	74.0	51.3	70.7	58.0	72.7	69.3	53.3	8.7
Gemini 3.5 Flash	62.7	64.0	56.7	60.7	52.0	57.3	58.7	66.0	69.3	64.7	4.9
Claude Opus 4.8	59.3	68.7	50.0	68.7	52.7	52.7	57.3	61.3	66.7	64.7	6.6
σ_{model}	4.2	3.3	2.9	5.5	0.5	7.6	0.5	4.7	1.3	5.3	–

Table 4: Per-language pass rates. σ_{lang} and σ_{model} are the standard deviations (in percentage points) across languages and across models, respectively.

Model	International-ization	Interaction w/ Environment	Text Conversion	Text Handling	Cultural & Other
GPT-5.5	63.5	70.4	71.3	53.3	58.0
Gemini 3.5 Flash	60.4	69.6	69.7	55.6	55.7
Claude Opus 4.8	60.8	64.8	65.0	53.3	57.2

Table 5: Per-category pass rates for the top three models.

Difficulty Calibration We used the same trial results to estimate task difficulty following Merrill et al. (2026). For each task, we pooled its 30 trial outcomes across the six models and computed the overall pass rate. We classified tasks with pass rates of at least 66.7% as easy, rates from 33.3% to below 66.7% as medium, and rates below 33.3% as hard. Table 3 shows an approximate 4:3:3 ratio of easy, medium, and hard tasks.

4.2 Performance Breakdown

Terminal-Bench-LILT enables analysis of model weaknesses by language and task category. These comparisons are descriptive rather than controlled: empirical difficulty varies across languages, and category sizes are unequal. Even with these limitations, the results reveal broad performance patterns worth investigating.

Per-Language Results Table 4 reports per-language pass rates for the top three models. Performance varies substantially across languages, as do the model rankings.

Hindi and German are the two lowest-scoring languages for every model, both around 50–53%. The two leading models show different language strengths: GPT-5.5 leads on Arabic, Spanish, Japanese, and Serbian, whereas Gemini 3.5 Flash leads on German and Korean. The largest gaps between them occur in Japanese (13.4 points), Spanish (13.3 points), and Chinese (11.4 points). Claude Opus 4.8 generally falls behind these two models but leads on Czech and Hindi.

Note that Gemini 3.5 Flash shows notably more uniform performance across languages than the others ($\sigma_{\text{lang}} = 4.9$), with no language below 52%. Among the ten languages, Hindi and Korean show the most consistent performance across models ($\sigma_{\text{model}} = 0.5$), whereas Japanese, Spanish, and Chinese show the largest cross-model variation.

Per-Category Results We also report pass rates for the top models by task category (Table 5). Text handling is the hardest category for every model, whereas text conversion and

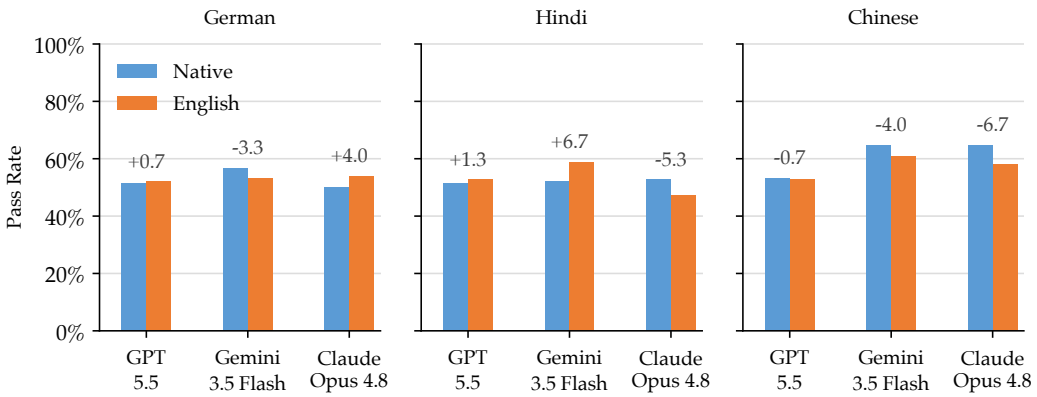


Figure 6: Native language vs. English instruction ablation. Every bar pair covers the same 30 tasks, scored in both conditions. The number above each pair is the difference in pass rate, English minus native.

environment interaction are the easiest. The gap between a model's best and worst category can be wide, reaching 18.0 points for GPT-5.5 between text conversion and text handling.

We hypothesize that this gap reflects how much language-specific knowledge each category requires. For example, text conversion relies on standardized transformations with direct library support (e.g., Unicode normalization), so a general solution often transfers across languages. In contrast, text handling depends on locale-specific rules that default tools handle incorrectly, such as Turkish case folding or stemming for agglutinative languages, forcing the model to supply the exact knowledge these tasks withhold.

4.3 Ablation Studies

Effect of Instruction Language Every Terminal-Bench-LILT task is accompanied by an English translation of its native instruction, which lets us run the same task with the translated instruction while holding the data, environment, and verifier fixed. This isolates the effect of switching the instruction language, comparing the target language against the models' strongest language, English.

Figure 6 shows this comparison in three languages with different scripts, using the top three models from the main results (Section 4.1). Across the nine model-language configurations, English instructions never move the pass rate by more than 7 points, and move it by less than 5 in six of them, similar to the small, inconsistent comprehension effects reported for translated benchmarks (Section 2). Even with English instructions, every configuration stays around 50–60%. This confirms the premise behind our native-first task design: instruction language is not the bottleneck; the hard part is working with native-script data under locale-specific rules.

Effect of Reasoning All LLMs tested in this work are capable of reasoning via chain-of-thought (Wei et al., 2022), which may improve problem solving on coding tasks at the cost of more output tokens and longer runtimes. To study this effect, we ran the evaluation with reasoning switched on and off and compared the results.[1] For this ablation, we effectively removed the tasks' time limit, letting each agent run until it finished organically. This study was run on 19 Terminal-Bench-LILT tasks that are hard under the empirical difficulty classification (Table 3). For comparison, we also ran the same reasoning ablation on 19 randomly sampled hard tasks from the original, English-only Terminal-Bench (Merrill et al.,

[1]When on, we used each provider's default reasoning level: *medium* for GPT-5.5 and Gemini 3.5 Flash, *high* for Claude Opus 4.8. Gemini 3.5 Flash cannot disable reasoning entirely by design, so for the off condition we set it to its *minimal* level.

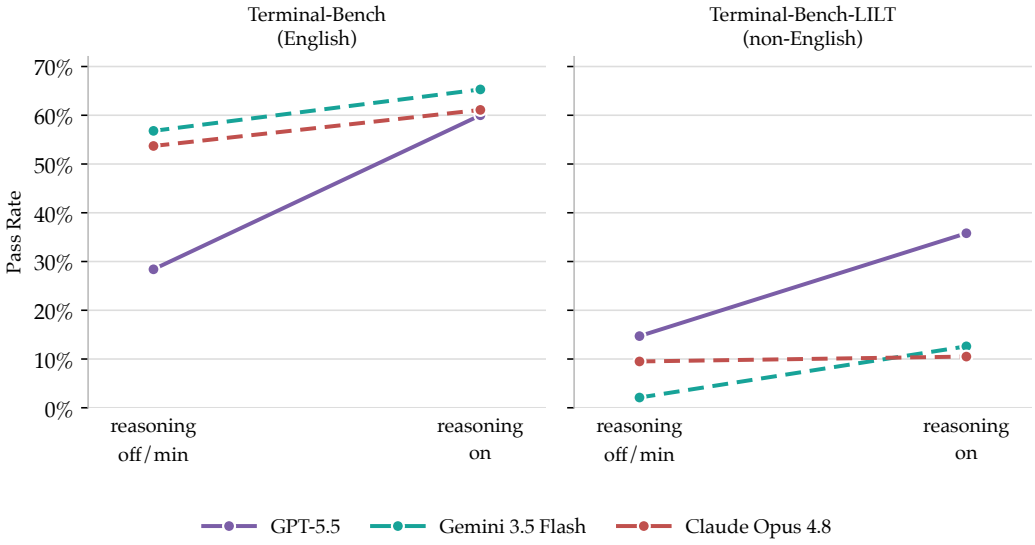


Figure 7: Reasoning ablation. Solid lines mark significant changes with $p < .05$ in a two-sided paired permutation test.

2026); note that their tasks pose pure engineering challenges, unlike ours, which also require linguistic and cultural knowledge.

Figure 7 shows the results. While reasoning improves GPT-5.5, no significant effect can be observed with Gemini 3.5 Flash or Claude Opus 4.8. With reasoning on, the three models converge to a similar level on the English tasks (60.0–65.3%), whereas on Terminal-Bench-LILT they range from 10.5% to 35.8%. Reasoning thus lifts English performance to a shared level, but leaves large differences in multilingual capability across models. Across models, reasoning inflates output token counts to 1.4–2.7× their off/minimal-reasoning levels, yet the large majority of the Terminal-Bench-LILT trials still fail. Note that some tasks in German and Hindi stay at a 0% pass rate even with reasoning.

5 Conclusion

We introduce Terminal-Bench-LILT, an agentic coding benchmark of 300 tasks written by native-speaker programmers in ten non-English languages. Each task requires language-, region-, and culture-specific knowledge that a native developer takes for granted, built into everyday engineering work rather than tested in isolation. A layered review pipeline retains only tasks that are realistic, verifiable, and genuinely locale-specific. The benchmark is far from saturated: frontier models still fail over a third of trials, many tasks go unsolved entirely, pass rates vary widely by language, and rankings do not follow general coding-benchmark standing. Our ablation studies show that this difficulty is not an artifact of non-English prompting or of limited compute. Terminal-Bench-LILT reveals gaps in multilingual capability across models, languages, and task categories that English benchmarks cannot expose. Its native-first design also offers a template for evaluation beyond English.

Ethics Statement

First, all tasks were written by native-speaker programmers who contributed voluntarily with informed consent and compensation; we screened submissions for personal, proprietary, or sensitive data and collected no identifying information about contributors. Second, our ten languages are a small, uneven sample and any single language spans many locales we do not fully cover, so low scores signal a measurement gap rather than a verdict on any language or its speakers. Third, this is a diagnostic benchmark for a narrow capability:

strong scores do not certify a model as fit for deployment, and weak scores should prompt further study, not exclusion of a language. Fourth, we embed canary strings to limit training-data contamination and plan to refresh the pool over time, and because these tasks execute code, all evaluation runs in sandboxed containers.

References

Anthropic. Claude Opus 4.7 System Card. Technical report, Anthropic, April 2026a.

Anthropic. Claude Opus 4.8 System Card. Technical report, Anthropic, May 2026b.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. Multi-lingual evaluation of code generation models. In *The Eleventh International Conference on Learning Representations (ICLR)*, 2023. arXiv preprint arXiv:2210.14868.

Josh Barua, Seun Eisape, Kayo Yin, and Alane Suhr. Long chain-of-thought reasoning across languages. In *The Fourteenth International Conference on Learning Representations (ICLR)*, 2026. arXiv preprint arXiv:2508.14828.

Masudul Hasan Masud Bhuiyan, Manish Kumar Bala Kumar, and Cristian-Alexandru Staicu. "write in English, nobody understands your language here": A study of non-English trends in open-source repositories. In *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering (ICSE '26)*, 2026. doi: 10.1145/3744916.3787766. arXiv preprint arXiv:2602.19446.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691, 2023. doi: 10.1109/TSE.2023.3267446. arXiv preprint arXiv:2208.08227.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

GitHub. Octoverse: A new developer joins GitHub every second as AI leads TypeScript to #1. The GitHub Blog, October 2025a.

GitHub. Application card: GitHub Copilot Agents. GitHub Docs, 2025b.

Google DeepMind. Gemini 3.1 Pro: Model card. Technical report, Google DeepMind, February 2026a.

Google DeepMind. Gemini 3.5 Flash: Model card. Technical report, Google DeepMind, May 2026b.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments. Zenodo, 2026. Software, version v0.16.1. doi: 10.5281/zenodo.20953922.

Omer Hofman, Jonathan Brokman, Oren Rachmil, Shamik Bose, Vikas Pahuja, Toshiya Shimizu, Trisha Starostina, Kelly Marchisio, Seraphina Goldfarb-Tarrant, and Roman Vainshtein. MAPS: A Multilingual Benchmark for Agent Performance and Security. *arXiv preprint arXiv:2505.15935*, 2025.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025. arXiv preprint arXiv:2403.07974.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024. arXiv preprint arXiv:2310.06770.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 TB of permissively licensed source code. *Transactions on Machine Learning Research*, 2023. arXiv preprint arXiv:2211.15533.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations (ICLR)*, 2026. arXiv preprint arXiv:2601.11868.

Junho Myung, Nayeon Lee, Yi Zhou, Jiho Jin, Rifki Afina Putri, Dimosthenis Antypas, Hsuvas Borkakoty, Eunsu Kim, Carla Perez-Almendros, Abinew Ali Ayele, Víctor Gutiérrez-Basulto, Yazmín Ibáñez-García, Hwaran Lee, Shamsuddeen Hassan Muhammad, Kiwoong Park, Anar Sabuhi Rzayev, Nina White, Seid Muhie Yimam, Mohammad Taher Pilehvar, Nedjma Ousidhoum, Jose Camacho-Collados, and Alice Oh. BLEnD: A benchmark for LLMs on everyday knowledge in diverse cultures and languages. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024) Datasets and Benchmarks Track*, 2024. arXiv preprint arXiv:2406.09948.

Tarek Naous, Michael J Ryan, Alan Ritter, and Wei Xu. Having beer after prayer? measuring cultural bias in large language models. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16366–16393, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.862.

Yuha Nishigata, Waka Ito, and Kimio Kuramitsu. Non-English code generation with cross-lingual chain of thought. In Chu-Ren Huang, Yasunari Harada, Jong-Bok Kim, Nguyen T.M. Huyen, Le Thanh Huong, Pham Hien, Emmanuele Chersoni, Le Minh

Nguyen, Rachel Edita Oñate Roxas, and Sherly Dita (eds.), *Proceedings of the 39th Pacific Asia Conference on Language, Information and Computation*, pp. 438–446, Hanoi, Vietnam, December 2025. Association for Computational Linguistics.

OpenAI. GPT-5.4 Thinking System Card. Technical report, OpenAI, March 2026a.

OpenAI. GPT-5.5 System Card. Technical report, OpenAI, April 2026b.

Qiwei Peng, Yekun Chai, and Xuhong Li. HumanEval-XL: A multilingual code generation benchmark for cross-lingual natural language generalization. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING)*, 2024. arXiv preprint arXiv:2402.16694.

Nishat Raihan, Antonios Anastasopoulos, and Marcos Zampieri. mHumanEval – a multilingual benchmark to evaluate large language models for code generation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 11432–11461, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.naacl-long.570.

Lisa Schut, Yarin Gal, and Sebastian Farquhar. Do multilingual LLMs think in English? In *ICLR 2025 Workshop on Building Trust in LLMs and LLM Applications*, 2025. arXiv preprint arXiv:2502.15603.

Siqi Shen, Lajanugen Logeswaran, Moontae Lee, Honglak Lee, Soujanya Poria, and Rada Mihalcea. Understanding the capabilities and limitations of large language models for cultural commonsense. In Kevin Duh, Helena Gomez, and Steven Bethard (eds.), *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 5668–5680, Mexico City, Mexico, June 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.naacl-long.316.

SlashData. There are 47.2 million developers in the world – Global developer population trends 2025. SlashData blog, May 2025.

Stack Overflow. Can I ask a question in a language other than English? Stack Overflow Help Center, 2024.

Chaozheng Wang, Zongjie Li, Cuiyun Gao, Wenxuan Wang, Ting Peng, Hailiang Huang, Yuetang Deng, Shuai Wang, and Michael R. Lyu. Exploring multi-lingual bias of large code models in code generation. *arXiv preprint arXiv:2404.19368*, 2024.

Zhiruo Wang, Grace Cuenca, Shuyan Zhou, Frank F. Xu, and Graham Neubig. MCoNaLa: A benchmark for code generation from multiple natural languages. In *Findings of the Association for Computational Linguistics: EACL 2023*, pp. 265–273, Dubrovnik, Croatia, May 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-eacl.20.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pp. 24824–24837, 2022.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*, 2025.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 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 (KDD)*, pp. 5673–5684, 2023. doi: 10.1145/3580305.3599790.

A Sample Tasks

Table 6: Sample tasks in the benchmark.

Task ID	Category	Description	Difficulty
`cs-implicit-alphabet-frequency`	Cultural & Other	Count occurrences of each Czech alphabet letter in a text without being told what the Czech alphabet is — the agent must implicitly know all 42 letters including accented characters and the 'ch' digraph.	Medium
`cs-nbsp-justify`	Cultural & Other	Split Czech sentences into xelatex-justified lines with no Overfull hbox, while honoring Czech orthotypographic non-breaking-space and soft-hyphen rules.	Hard
`cs-rozuctovani-tepla`	Cultural & Other	Fix a script that audits a Czech SVJ heating-cost allocation against vyhlaska 269/2015 Sb.	Medium
`de-bgb-citation-resolver`	Cultural & Other	Resolve a list of German civil-code citations (e.g. § 433 Abs. 1 Satz 2) against a BGB XML file. Parse the hierarchical legal-text format, count Sätze per Absatz while respecting period-terminated legal abbreviations (Abs., Nr., S., gem., i.V.m.), and return each citation's exact text as JSON.	Medium
`de-umsatzsteuer-klassifikation`	Cultural & Other	Classify German products by statutory VAT rate (7% reduced or 19% standard) under §12 UStG and Anlage 2 (Stand 1.1.2026), fixing a broken keyword draft. Many names are German compounds where the Grundwort sets the rate while the Bestimmungswort misleads, so decompose the compound first.	Hard
`es-locale-search-analyzer`	Text Handling	Build a Spanish search analyzer that indexes a corpus, answers queries with locale-correct collation, and emits snippets that preserve the original surface form.	Easy
`ja-terminal-width`	Internationalization	Wrap each line of a NOC-log file to a fixed 40-column width: strip ANSI escapes, fold halfwidth katakana to fullwidth (merging ˚ °), measure width (CJK/fullwidth/emoji = 2, ASCII = 1, variation-selector clusters = one 2-wide unit), never splitting a wide char. Bytes stay verbatim, so no NFC/NFKC.	Medium
`sr-cadastral-parcel-ledger`	Text Conversion	Normalize Serbian cadastral parcel request text across registry aliases, script variants, area units, ownership shares, and Serbian collation into a JSON ledger.	Easy
`sr-locale-sort`	Cultural & Other	Fix a buggy Serbian Latin sorter so the digraphs dž, lj and nj are treated as single letters in any word position, not only at word start.	Easy
`tr-username-dedup-casefold`	Text Handling	Detect duplicate user registrations in a Turkish CRM by applying the Turkish-locale-aware username canonicalization (NFC + I-pair pre-fold + casefold) that Python's str.lower() and str.casefold() do not implement by default.	Hard