Beyond Accuracy: Realistic and Diagnostic Evaluation of Code Generation Models

Pareesa Ameneh Golnari[*,1], **Adarsh Kumarappan**[*,2], **Wen Wen**[1], **Xiaoyu Liu**[1], **Gabriel Ryan**[1], **Yuting Sun**[1], **Shengyu Fu**[1], **Elsie Nallipogu**[1]
[1]Microsoft, [2]California Institute of Technology
`{pgolnar, wenwen, lixiaoyu, ryangabriel, yutingsun, shengyfu, elsien}@microsoft.com, adarsh@caltech.edu`

Abstract

DevBench is a telemetry-driven benchmark designed to evaluate Large Language Models (LLMs) on realistic code completion tasks. It includes 1,800 evaluation instances across six programming languages and six task categories derived from real developer telemetry, such as API usage and code purpose understanding. Unlike prior benchmarks, it emphasizes ecological validity, avoids training data contamination, and enables detailed diagnostics. The evaluation combines functional correctness, similarity-based metrics, and LLM-judge assessments focused on usefulness and contextual relevance. 11 state-of-the-art models were assessed, revealing differences in syntactic precision, semantic reasoning, and practical utility. Our benchmark provides actionable insights to guide model selection and improvement—detail that is often missing from other benchmarks but is essential for both practical deployment and targeted model development.

1 Introduction

Large Language Models (LLMs) have transformed modern software development by enabling advanced code generation, powering tools like GitHub Copilot [GitHub, 2025] and Cursor [AnySphere, 2025]. As these systems are increasingly integrated into real-world workflows, realistic and rigorous evaluation frameworks are essential to understanding their strengths and limitations.

Several types of benchmarks have been proposed to evaluate different aspects of LLM code generation. *Problem solving benchmarks* focus on writing solutions to coding problems, either manually written [Chen et al., 2021, Liu et al., 2023, Austin et al., 2021] or collected from coding challenge websites [Hendrycks et al., 2021]. *Repository-based benchmarks* target challenges in large open-source projects, such as reasoning across multiple files and external APIs [Wu et al., 2024, Ding et al., 2023, Li et al., 2024b, Zhuo et al., 2025]. *Evolving benchmarks* seek to address training data contamination by updating and segmenting their evaluation instances based on repository updates [Li et al., 2024a, Jain et al., 2024].

However, existing benchmarks rely on code samples scraped from open source repositories or coding challenge websites and generate target completions based on static rules for filling in line, function, or class implementations. This limits them in several ways: First, the target completions are not based on real world usage patterns for code completion tools, and therefore do not focus on common challenging completion scenarios that arise in real world usage. Second, the diagnostic value of these benchmarks is limited because they report aggregate metrics, but cannot attribute differences in performance to specific usage areas. Third, benchmarks collected from publicly available sources

[*]Equal contribution.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Deep Learning for Code (DL4C) Workshop.

Table 1: Comparison of a collection of recent code generation benchmarks across size, language coverage, focus, source, and unique features.

Benchmark	# Tasks	Languages	Focus	Source	Unique Feature
RepoMasterEval	288	Python, TypeScript	Real-world repository completion	GitHub repos (>100 stars)	Mutation testing for test robustness
CrossCodeEval	∼10k	Python, Java, TypeScript, C#	Cross-file dependencies	GitHub repos (>3 stars)	Static analysis for dependencies
CoderEval	460	Python, Java	Cross-file pragmatic generation	GitHub repos (popular tags)	Human-labeled docstrings
ClassEval	100	Python	Class-level generation	Manually crafted	Multiple interdependent methods
HumanEval	164	Python	Basic programming tasks	Manually crafted	Simple interview-style problems
HumanEval+	164	Python	Enhanced testing rigor	Manually crafted	80× more evaluation instances
LiveCodeBench	511	Python	Contamination-free evaluation	Competition platforms	Time-based contamination tracking
SWE-bench	2,294	Python	Repository-level bug fixing	GitHub issues and PRs	Real-world issues from 12 popular repos
BigCodeBench	1,140	Python	Diverse function calls as tools	Human-LLM collaborative generation	723 function calls from 139 libraries across 7 domains
DevBench (this work)	**1,800**	**Python, JavaScript, TypeScript, Java, C++, C#**	**Realistic developer-informed scenarios**	**Synthetically generated, manually reviewed**	**Telemetry-guided, human-validated**

are prone to training data contamination, which has been observed in models overfitting to existing benchmarks [Jain et al., 2024].

In this work, we introduce **DevBench**, a realistic and scalable benchmark grounded in observed developer behavior. **DevBench** focuses on common yet challenging completion scenarios, identified from internal telemetry and synthesized into 1,800 evaluation instances spanning six languages and six task categories. Each instance is reviewed for quality and realism, ensuring that tasks reflect how developers actually use code completion tools while remaining contamination-resistant.

As shown in Table 1, **DevBench** advances beyond existing benchmarks in both realism [Paul et al., 2024] and scope. It offers four key advantages: **(1) realism**, with tasks rooted in observed developer behavior; **(2) contamination resistance**, through synthetic but controlled instance generation; **(3) fine-grained evaluation**, assessing semantic alignment and developer utility; and **(4) cross-language coverage**, spanning Python (Py), JavaScript (JS), TypeScript (TS), Java, C++, and C#.

Together, these features provide ecological validity: **DevBench** reflects **authentic developer challenges** rather than hypothetical tasks, is validated through expert review and organizational dogfooding, and captures diverse contexts across languages and developer skill levels. By enabling both overall rankings and scenario-specific diagnostics, **DevBench** supports informed model selection and optimization, and provides a contamination-resilient foundation for future research.

2 Benchmark Design

We introduce **DevBench**, a realistic benchmark grounded in large-scale developer telemetry. We view code generation as a composite, puzzle-solving task in which models must combine distinct capabilities, such as API usage, intent understanding, and syntax control. To evaluate these skills, we define benchmark categories that isolate each capability while ensuring every instance is solvable from the provided prefix/suffix, making evaluation both realistic and fair. Although individual instances are synthesized, **DevBench** is telemetry-driven: its categories, task types, and scenarios are derived from analysis of over one billion real developer interactions, with synthesis used only to instantiate these empirically derived patterns in a privacy-preserving, contamination-resistant manner.

2.1 From User Telemetry to Categories

The benchmark categories are derived from an internal telemetry dataset containing over one billion anonymized code completions, each recording the prefix, suffix, generated and golden completions,

Table 2: Language-specific adaptations. Here, ML = Machine Learning, HOFs = Higher-Order Functions, RAII = Resource Acquisition Is Initialization, and HW accel. = Hardware Acceleration.

Category	Python	C#	C++	Java	JavaScript	TypeScript
API Usage	ML libs, scientific computing	.NET Core, ASP.NET	Systems prog., graphics, HW accel.	JDK, enterprise frameworks	Browser APIs, Node.js modules	Same as JS w/ types
Code Purpose	Iterators/generators, context mgrs	LINQ/collections, async-await	Iterators/algorithms, multithreading	Streams/collections, lambdas	Closures, promises/async	Same as JS w/ type systems
Code2NL	Docstrings	XML docs	Doxygen comments	Javadoc	JSDoc	TSDoc w/ type annot.
Low Context	Decorators, context mgrs	Complex generics, LINQ	Template metaprog., RAII	Lambda expr., streams	Async patterns, HOFs	Adv. type features
Pattern Matching	Decorators, context mgrs	Generic prog., memory mgmt	Template metaprog., algorithms	Streams, HOFs	Async, event handling	Type defs, generics
Syntax Completion	Decorator stacking, nested contexts	LINQ expr., generic constraints	Template metaprog., RAII patterns	Stream ops, try-with-resources	Promise chaining, generators	Interface defs, adv. types

and user interactions (accept, reject, edit). This dataset spans diverse contexts over IDEs, geographical locations, language distribution and developers range from students to senior engineers, across both individual and enterprise environments.

To meet privacy and compliance requirements, we do not use raw user code. Instead, we construct synthetic evaluation instances that replicate the structural complexity and usage patterns observed in telemetry. Completions were sampled and annotated to identify common failure modes, bottlenecks, and characteristic prompt structures. These findings were refined through iterative discussions with a research group that included language specialists, ensuring that the resulting categories captured both statistical prevalence and realistic, high-impact developer workflows.

We further incorporated feedback from internal dogfooding and public user reports to strengthen ecological validity. Reviewed samples were checked for representativeness so that the resulting benchmark categories capture common scenarios with clear evaluation criteria, edge cases, and realistic challenge levels directly grounded in developer behavior.

2.2 Benchmark categories

We define six benchmark categories based on our analysis of user telemetry. Each category targets a distinct type of developer intent and is consistently evaluated across languages, with adaptations to reflect the idioms and ecosystems of each target language (see Table 2). The categories are described in detail below (examples in Appendix A).

API Usage: This category tests a model's ability to correctly apply specialized library functions. Each evaluation instance consists of a prefix that sets the context, a golden completion illustrating proper API usage, and a suffix for continuation.

Code Purpose Understanding: This category evaluates whether a model can generate code that aligns with the underlying business logic and domain-specific conventions—not just syntactic correctness. For instance, consider a `BankAccount` class where a `withdraw` method is already implemented. We prompt the generative model to implement a new `transfer` method. Based on the existing code, the model is expected to infer the intended functionality of the new method, reuse the existing `withdraw` logic for consistency, and ensure that the amount is positive and sufficient funds are available. This task goes beyond syntactic correctness, evaluating the model's ability to reason about object-oriented design and domain-specific financial logic.

Code2NL/NL2Code: This category evaluates a model's ability to translate between code and natural language (NL) in both directions. This reflects real-world developer workflows, where boundaries between code and language are increasingly blurred. To align with practical use cases, our benchmark covers a wide spectrum of scenarios including: (1) NL only in the prefix, (2) code only in the prefix, (3) mixed NL and code in the prefix, (4) various NL forms including docstrings, inline comments, block comments, and user-facing documentation, and (5) different documentation styles across programming languages (e.g., Python docstrings, JavaDoc, JSDoc, XML docs, and Doxygen).

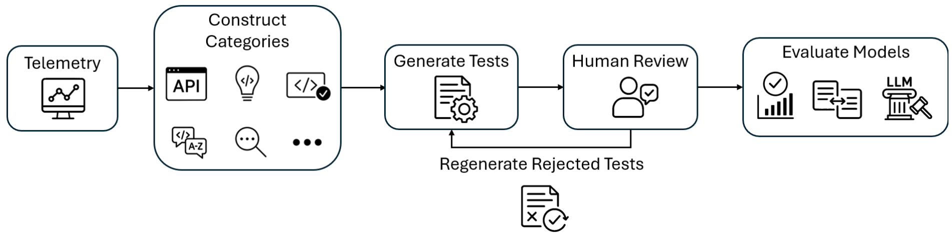

Figure 1: This diagram presents the end-to-end **DevBench** pipeline, starting with developer telemetry analysis to define six code completion categories. Evaluation instances are synthetically generated and refined through human review. Final evaluation combines functional correctness, similarity-based metrics, and LLM-based judgment to assess functional, semantic, and holistic model performance.

Low Context: This category evaluates a model's ability to complete code using minimal context (10–20 lines total), requiring it to recognize language-specific patterns and idioms. These tasks are carefully designed to be solvable despite limited information, testing the model's deep understanding of programming conventions without relying on broader context.

Pattern Matching: This category tests a model's ability to recognize and extend established code patterns within realistic contexts. Each test includes 2–3 clear examples in the prefix, combining a technical pattern (e.g., error handling) with a domain context (e.g., security), ensuring the model must follow the intended structure rather than generating arbitrary code.

Syntax Completion: This category tests a model's ability to generate complex, nested structures while adhering to language-specific syntax rules. Evaluation instances span four categories: nested control structures, complex features, multi-line patterns, and error handling. The model must correctly manage indentation, close code blocks, and match braces or parentheses, demonstrating mastery of each language's unique syntactic constructs.

2.3 Benchmark construction

DevBench covers key scenarios observed in real developer interactions, including both prefix-only completions and fill-in-the-middle (FIM) cases where a suffix is provided. Completions may begin at arbitrary points within a sentence, reflecting practical usage patterns. Each evaluation instance consists of four components: (1) a **prefix** providing the preceding code context, (2) a **completion** as the expected model output, (3) a **suffix** representing subsequent code, and (4) **assertions** to validate correctness. For Java, C#, and C++, assertions are embedded in the suffix, while for Python, JavaScript, and TypeScript, they are placed in a separate section. This design supports evaluation of both syntactic and semantic correctness in a manner consistent with real coding workflows.

Figure 1 summarizes the construction pipeline. Synthetic instances were generated with OpenAI's GPT-4o, chosen for its fluency, reasoning, and code generation capabilities [OpenAI et al., 2024]. Recent studies suggest GPT-4o introduces minimal stylistic bias [Maheshwari et al., 2024, Chen et al., 2024], and our human validation further mitigates risks; notably, multiple non-GPT models (e.g., Claude 3.5 Sonnet, DeepSeek-V3) outperform GPT-4o on **DevBench** (Section 4). Generation used temperature 0.7 with a 4000-token limit.

Each synthetic instance was first screened via automatic syntax checks, then reviewed manually with a custom annotation tool displaying prefix, completion, and suffix together. Annotators evaluated three dimensions: (1) *usefulness* (does the completion satisfy a plausible developer need?), (2) *realism* (does it fit real-world workflows?), and (3) *category alignment* (is it consistent with the intended task type?). For example, API Usage cases were checked for correct library calls, while Low Context cases were validated for minimal but sufficient context.

Rejected samples—typically due to low challenge or category mismatch—were regenerated and re-verified. This hybrid process yielded 1,800 high-quality evaluation instances (50 per category per language), each grounded in telemetry-derived patterns yet privacy-preserving and contamination-resistant. All instances are executable with assertions, enabling assessment of both syntax and functionality, though execution environments are fully implemented only for Python at present.

Table 3: Complexity of the benchmarks

Metric	**DevBench (ours)**	CrossCodeEval	CoderEval-Py	CoderEval-Java	APPS	HumanEval	MBPP	Concode	CoNaLA	DS-1000
Avg. LOC	61.8	71.1–116.5	32.0	10.2	21.4	11.5	6.8	4.8	1.0	3.8
Cyclomatic Complexity	4.9	–	4.7	3.1	–	3.6	–	1.4	–	–

Table 4: DevBench language-specific statistics

Language	Prefix LOC	Completion LOC	Total LOC	Prefix Tokens	Completion Tokens	Cyclomatic
Python	19.5	4.2	40.4	92.5	21.8	2.2
C#	47.2	4.7	67.8	181.1	30.3	4.2
C++	42.0	5.1	60.9	194.2	39.4	5.7
Java	34.7	4.9	55.3	153.7	36.1	4.6
JavaScript	40.1	5.9	69.9	217.9	43.1	6.2
TypeScript	49.6	6.9	76.6	282.6	52.4	6.5
Average	**38.8**	**5.3**	**61.8**	**187.0**	**37.2**	**4.9**

Complexity. To assess the complexity of DevBench, we report *lines of code (LOC)*, *token counts*, and *cyclomatic complexity* [Landman et al., 2016]. As shown in Table 3 and Table 4, DevBench offers higher complexity and realism than prior benchmarks, with evaluation instances averaging 61.8 LOC and a cyclomatic complexity of 4.9. Importantly, DevBench maintains a balanced prefix-to-completion ratio: completions average 5.3 LOC, with 187.0 tokens in the prefix and 37.2 in the completion. In contrast, CrossCodeEval features long prompts (71–116 LOC) but extremely short completions (1–2 LOC). This balance makes DevBench more reflective of practical code-completion workflows, where both context and generated code contribute meaningfully to task complexity.

3 Evaluation methods

Given the challenges in evaluating LLMs, we employ a combination of methods: functionality correctness; similarity-based metrics, which offer fast, scalable evaluation across languages; and LLM-judge evaluations to assess output quality from a human-aligned perspective.

3.1 Functional correctness

For functionality correctness, we report Pass@1 (single-attempt success rate) [Chen et al., 2021]. Currently, execution-based evaluation is fully implemented for Python evaluation instances. For the remaining five languages, we ensure golden completion reliability through automatic syntax validation using language-specific compilers, expert review to verify assertion satisfaction, and spot-check execution testing.

3.2 Similarity-Based evaluation

To capture complementary aspects of model performance, we use two widely adopted similarity metrics: Average Cosine Similarity and Line 0 Exact Match Rate. Average Cosine Similarity assesses semantic equivalence across the full completion, even when syntax differs, while Line 0 Exact Match focuses on strict precision at the start of the completion.

Average Cosine Similarity: We use token-based cosine similarity [Zhou et al., 2023] to measure semantic overlap between the model-generated and golden completions. Formally, we tokenize both texts and compute:

$$\text{CosineSimilarity}(A, B) = \frac{A \cdot B}{||A|| \cdot ||B||} \tag{1}$$

where A and B are token count vectors. When tokenization fails due to unusual code constructs, we fall back to character n-grams (1-3) to ensure robust comparison across all samples.

Line 0 Exact Match Rate: We calculate the Exact Match Rate [Ding et al., 2023] between the first lines of model-generated and golden completions. This strict metric measures the percentage of cases where the model produces exactly the same first line as the reference solution:

$$\text{Line0ExactMatch}(A, B) = \begin{cases} 1, & \text{if } A_{\text{line0}} = B_{\text{line0}} \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

where A_{line0} and B_{line0} are the first lines of the model-generated and golden completions, respectively.

3.3 LLM-judge evaluation

For automated code quality evaluation, we designed an LLM-based judge that scores each completion along two dimensions: **relevance** to the provided context and **helpfulness** in advancing the task. These dimensions were chosen because they correlate strongly with developer telemetry signals (accept/reject/edit) and reflect how practitioners evaluate code in practice. Each aspect is rated on a 0–5 scale, yielding a combined score from 0 to 10.

Bias Mitigation. We employ `o3-mini` as the judge model, selected for its favorable bias profile: according to the OpenAI System Card, it shows the lowest measured bias among comparable models on discrimination tasks [Tong and Zhang, 2024]. To align evaluation with real developer preferences, we calibrated the judge prompt against acceptance signals in telemetry data, refining both phrasing and rating granularity to emphasize *practical utility* over superficial similarity.

To further reduce evaluation bias, the judge is never informed of the identity or architecture of the model producing the completion. Empirically, `o3-mini` does not display self-preference bias—other models (e.g., Claude 3.5 Sonnet, DeepSeek-R1) achieve higher judged scores in our experiments (Section 4)—indicating that rankings are driven by code quality rather than architectural affinity.

Confidence Interval. For each model, we compute average scores by programming language and evaluation scenario. We then aggregate completions within each language to obtain an overall average and a 95% confidence interval, estimated via 10,000 bootstrap resamples. Finally, we report the overall average score across all languages and completions with its corresponding confidence interval.

4 Experiments

4.1 Experimental setup

We evaluated a diverse set of state-of-the-art language models to capture code generation performance across varying training approaches and scales. Our selection includes multiple OpenAI models (e.g., GPT-4o, GPT-4o mini [OpenAI et al., 2024], GPT-4.1 mini [OpenAI et al., 2025a], GPT-4.1 nano [OpenAI et al., 2025a], and o3-mini [OpenAI et al., 2025b]), Anthropic's Claude 3.5 [Anthropic, 2024] and 3.7 Sonnet [Anthropic, 2025], DeepSeek-R1 [DeepSeek-AI et al., 2025a] and V3 [DeepSeek-AI et al., 2025b], GitHub's GPT-4o Copilot (April 2025) [GitHub, 2025], and Ministral-3B [Mistral, 2025] as a representative compact open model.

We designed our evaluation to mirror realistic usage scenarios. Following prior work [Jain et al., 2024], we used a temperature of 0.2 for more deterministic completions and set a maximum output length of 800 tokens to accommodate complex completions without excessive inference cost. All models used temperature-controlled nucleus sampling with a top-p of 1.0, preserving the full token distribution while modulating randomness via temperature. For LLM-judge evaluation we used o3-mini as a strong reasoning model with default settings (temperature=1.0 and top-p=1.0).

Models were evaluated in a zero-shot setting, each prompted using a consistent, code-only template, excluding explanations or comments. For models with format-specific outputs (e.g., Claude's special tags), we applied post-processing to extract functional code for fair comparison. See Appendix E.3 for more prompt details.

Our benchmark generation and evaluation workloads were distributed across cloud-based model APIs and local computing resources. Model API calls were orchestrated from a standard laptop (11th Gen Intel i7-1165G7 @ 2.80GHz with 16GB RAM) running Python 3.10. For benchmark generation,

Table 5: Pass@1 rates across code completion categories. All results used a temperature of 0.2.

Model	Overall ↓	API Usage	Code Purpose	Code2NL/NL2Code	Low Context	Pattern Matching	Syntax
Claude 3.7 Sonnet	76.25%	80.00%	84.00%	66.00%	90.00%	68.00%	60.00%
Claude 3.5 Sonnet	75.31%	82.00%	84.00%	64.00%	90.00%	66.00%	62.00%
DeepSeek-V3	73.75%	78.00%	82.00%	62.00%	86.00%	62.00%	66.00%
GPT-4.1 mini	73.44%	76.00%	74.00%	64.00%	88.00%	60.00%	72.00%
GPT-4o	71.56%	74.00%	78.00%	62.00%	90.00%	54.00%	66.00%
o3-mini	70.62%	78.00%	76.00%	60.00%	80.00%	54.00%	70.00%
GPT-4o Copilot	69.69%	74.00%	74.00%	58.00%	88.00%	50.00%	68.00%
GPT-4o mini	68.12%	72.00%	70.00%	64.00%	86.00%	48.00%	66.00%
DeepSeek-R1	63.12%	74.00%	74.00%	54.00%	82.00%	42.00%	48.00%
GPT-4.1 nano	57.19%	58.00%	60.00%	50.00%	68.00%	42.00%	62.00%
Ministral 3B	41.25%	46.00%	44.00%	34.00%	56.00%	36.00%	24.00%

using the OpenAI API (GPT-4o) to create synthetic evaluation instances required approximately 2-5 hours of wall-clock time for all languages, depending on API latency and excluding human review time. Each individual model evaluation on the complete benchmark required approximately 1.5-3 hours of wall-clock time, also dependent on the API latency. The Python execution component of our evaluation pipeline, which verifies functional correctness, was executed on the same laptop and required approximately 15 minutes per model (details in Appendix E.1).

4.2 Results and insights

4.2.1 Functional correctness (Pass@1)

Table 5 shows Pass@1 results, i.e., the percentage of completions that pass all assertions on the first attempt across the Python cases, revealing several notable findings.

Top Performers: Claude 3.7 Sonnet (76.25%), Claude 3.5 Sonnet (75.31%), and DeepSeek-V3 (73.75%) achieved the highest overall performance.

Small-Size Models: Ministral-3B achieves a modest overall Pass@1 of 41.25%.

Syntax Completion: GPT-4.1 mini outperforms larger models in the *Syntax Completion* category (72% vs GPT-4o 66%), similarly DeepSeek-V3 outperforms in the same category (66%) relative to DeepSeek-R1 (48%). This suggests that syntactic completion capabilities may not strictly correlate with model size or overall performance.

Pattern Matching: The *Pattern Matching* category shows wide variance: top models reach 66–68%, whereas GPT-4.1 nano and Ministral-3B lag at 42% and 36%. This shows that the ability to identify and extend established patterns in code is a differentiator between candidate models.

API Usage: Claude 3.5 Sonnet leads with 82%, followed by Claude 3.7 Sonnet at 80%, while small models GPT-4.1 nano and Ministral-3B lag at 58% and 46%.

4.2.2 Similarity-Based evaluation

Table 6 reports similarity metrics across languages for all models evaluated in our benchmark. Due to space constraints, Figure 2 focuses on four representative models that capture the breadth of the current model landscape: o3-mini (OpenAI's reasoning-focused model), Claude 3.7 Sonnet (Anthropic's foundation model), DeepSeek-V3 (a leading open-source model), and GPT-4o (OpenAI's proprietary model). The complete set of similarity results is available in Appendix E.4. Key observations from Figure 2 and Table 6 are discussed below.

High Performers: Claude 3.7 Sonnet exhibits the lowest Average Cosine Similarity variance, indicating strong cross-language consistency. Similarly, DeepSeek-R1 shows a generalist profile with low Line 0 Exact Match Rate variance, suggesting stable performance across languages. GPT-4o and GPT-4o Copilot perform particularly well in Java and C++.

Most and Least Challenging Categories: In line with the Pass@1 results, the *Low Context* category consistently achieves the highest scores across models. Conversely, *Code2NL/NL2Code* remains the most difficult category, reinforcing our earlier observation that bidirectional translation between NL and code continues to pose a significant challenge for current models.

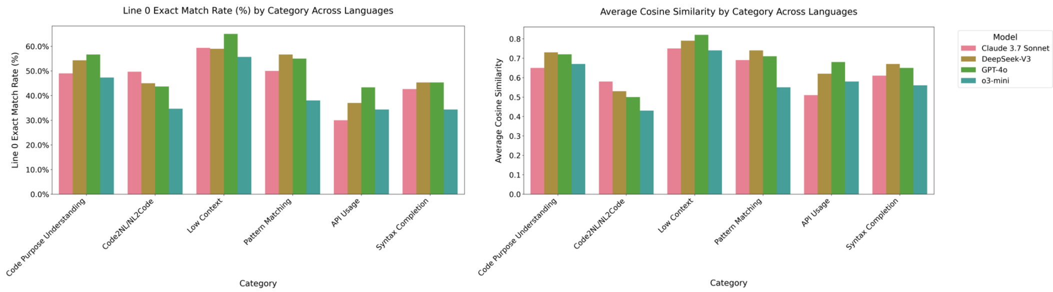

Figure 2: Similarity metrics by task category for representative models (all languages).

Table 6: Similarity metrics across programming languages. All results used a temperature of 0.2.

	Average Cosine Similarity						Line 0 Exact Match Rate (%)					
Model	Py	JS	TS	Java	C++	C#	Py	JS	TS	Java	C++	C#
Claude 3.7 Sonnet	0.65	0.60	0.50	0.70	0.68	0.66	44.67	44.67	34.33	55.0	54.33	47.67
Claude 3.5 Sonnet	0.70	0.59	0.54	0.77	0.76	0.69	49.33	42.0	37.33	60.0	60.33	49.0
DeepSeek-V3	0.70	0.62	0.58	0.76	0.75	0.68	49.0	43.67	39.67	61.33	59.0	44.67
GPT-4.1 mini	0.70	0.60	0.51	0.72	0.72	0.66	51.0	44.0	36.67	55.0	58.0	46.0
GPT-4o	0.69	0.63	0.56	0.76	0.75	0.69	48.67	47.67	37.67	62.33	61.67	51.0
o3-mini	0.62	0.53	0.45	0.64	0.69	0.59	43.67	35.0	26.0	48.67	54.67	36.33
GPT-4o Copilot	0.66	0.61	0.51	0.75	0.71	0.67	46.0	44.67	32.67	60.67	57.0	48.0
GPT-4o mini	0.64	0.58	0.50	0.70	0.70	0.66	43.33	40.0	30.33	54.33	56.33	42.67
DeepSeek-R1	0.68	0.60	0.55	0.68	0.73	0.65	44.33	40.67	34.0	52.67	54.33	43.0
GPT-4.1 nano	0.59	0.50	0.46	0.66	0.63	0.60	35.33	30.0	27.33	45.33	48.33	34.33
Ministral-3B	0.52	0.38	0.35	0.50	0.42	0.51	29.0	18.67	14.67	27.67	22.67	25.33

Metric Discrepancies: In Pattern Matching, Claude 3.7 Sonnet scores 68% on Pass@1 but only 50.0% on Line 0 Exact Match Rate and 0.69 in Average Cosine Similarity. In contrast, DeepSeek-V3 achieves higher similarity (56.67%, 0.74) but lower Pass@1 (62%). This discrepancy helps identify areas where deeper analysis is needed to understand model behavior. For example, DeepSeek-V3 reliably replicates familiar code patterns but sometimes fails to maintain full functional correctness, while Claude's solutions, while functionally correct, frequently employ alternative implementation approaches that diverge syntactically from the reference solutions which match patterns.

Challenging Language: TypeScript emerges as the most challenging language with an average performance drop of 23.9% compared to the average scores. This consistent difficulty stems from its complex type system and the need to maintain strict type consistency throughout the code.

Please see Appendix B and Appendix D for qualitative examples of the mentioned behaviors.

4.2.3 LLM-judge evaluation

Figure 3 presents the final LLM-judge scores with 95% confidence intervals and Appendix C provides a detailed breakdown by category and language. Unlike functionality- and similarity-based evaluations, the LLM-judge uses subjective, qualitative assessments, leading to rankings that may differ from those of other methods.

High Performers: Claude 3.5 Sonnet ranks highest according to the LLM-judge, followed closely by DeepSeek-R1 and o3-mini, revealing a ranking that differs from other evaluation methods.

Reasoning-Oriented Models: Compact, reasoning-focused models like o3-mini outperform larger, non-reasoning-centric models such as GPT-4o and Claude 3.7 Sonnet. These models likely benefit from chain-of-thought decomposition capabilities that enhance their ability to interpret and structure code completions in a contextually relevant manner.

Smaller Models: Smaller models tend to show wider confidence intervals, indicating greater variability and reduced consistency—particularly on complex or edge-case prompts—even when their average performance is competitive.

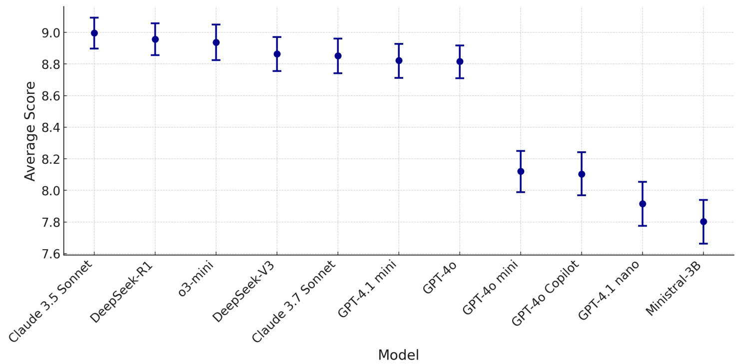

Figure 3: Final model scores with 95% confidence intervals, based on LLM-judge evaluation.

4.3 Diagnostic Case Study: DeepSeek-V3

DevBench's multi-metric framework enables fine-grained model diagnosis beyond aggregate rankings. Consider DeepSeek-V3: while achieving competitive overall performance, our analysis reveals specific improvement opportunities.

Syntax vs. Semantics: DeepSeek-V3 excels in Pattern Matching similarity in Table 8 (Average Cosine 74% vs. Claude 3.7 Sonnet's 69% and Line 0 Exact Match Rate 56.67% vs. 50%) but underperforms in functional correctness in Table 5 (62% vs. 68% Pass@1). This pattern indicates heavier reliance on pattern memorization than true semantic understanding. Manual review of failure cases confirms that DeepSeek-V3 often produces code syntactically close to the golden solution but functionally incorrect.

Category-Level Patterns: Based on Table 8, the model demonstrates strong performance in Pattern Matching (0.74 vs 0.69) and Syntax Completion (0.67 vs. 0.61) but underperforms in Code2NL/NL2Code tasks (0.53 vs. 0.58). This disparity reveals the model's tendency to memorize surface patterns rather than deeply understand and generate code in semantically rich tasks requiring bidirectional translation between natural language and code.

Language-Specific Gaps: While DeepSeek-V3 performs strongly in Python, it underperforms in C++ and Java, ranking 6th and 7th respectively (Table 7). Our cross-model analysis suggests potential gains, as structurally similar languages like C++ and Java often benefit from shared improvements.

Preserving Strengths: DeepSeek-V3 already excels in Syntax Completion and Python development, areas that should be maintained during future fine-tuning to avoid catastrophic forgetting.

These insights translate to actionable training priorities: (1) emphasize pattern extension and reasoning during fine-tuning to reduce over-reliance on memorization, (2) increase Code2NL/NL2Code training examples to improve semantic understanding, (3) include more Java and C++ samples in the training mix to close performance gaps, and (4) maintain current strengths in Python and Syntax Completion.

5 Conclusion

We introduced **DevBench**, a synthetic benchmark grounded in developer telemetry, enabling fine-grained, realistic code completion evaluation across six languages and task categories, resulting in 1,800 evaluation instances, with a focus on ecological validity, contamination resistance, and interpretability.

DevBench offers detailed insights into model behavior. Evaluating 11 state-of-the-art models, we observed consistent strengths in low-context pattern recognition and persistent challenges in bidirectional natural language–code translation and syntactic alignment. Our multi-pronged evaluation—combining functional correctness, similarity metrics, and LLM-judge assessments—revealed nuanced differences such as cross-language consistency and robustness across task types. By releasing the benchmark and its generation infrastructure, we aim to support the research community in advancing more accountable, targeted, and practical evaluation of code generation models.

References

Anthropic. Introducing Claude 3.5 Sonnet. https://www.anthropic.com/news/claude-3-5-sonnet, 2024.

Anthropic. Claude 3.7 Sonnet. https://www.anthropic.com/claude/sonnet, 2025.

AnySphere. Cursor - The AI Code Editor, 2025. URL `https://www.cursor.com/`.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, et al. Program Synthesis with Large Language Models, August 2021.

Hao Chen, Abdul Waheed, Xiang Li, Yidong Wang, Jindong Wang, Bhiksha Raj, and Marah I. Abdin. On the Diversity of Synthetic Data and its Impact on Training Large Language Models, October 2024.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, et al. Evaluating Large Language Models Trained on Code, July 2021.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, et al. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning, January 2025a.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, et al. DeepSeek-V3 Technical Report, February 2025b.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, et al. CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion, November 2023.

GitHub. GPT-4o Copilot: Your new code completion model is now generally available - GitHub Changelog. https://github.blog/changelog/2025-03-27-gpt-4o-copilot-your-new-code-completion-model-is-now-generally-available/, 2025.

GitHub. GitHub Copilot: Your AI pair programmer, 2025. URL `https://github.com/features/copilot`.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, et al. Measuring Coding Challenge Competence With APPS, November 2021.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, et al. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code, June 2024.

Davy Landman, Alexander Serebrenik, Eric Bouwers, and Jurgen J. Vinju. Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions. *Journal of Software: Evolution and Process*, 28(7):589–618, July 2016. ISSN 2047-7473, 2047-7481. doi: 10.1002/smr.1760.

Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. EvoCodeBench: An Evolving Code Generation Benchmark Aligned with Real-World Code Repositories, March 2024a.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, et al. DevEval: A Manually-Annotated Code Generation Benchmark Aligned with Real-World Code Repositories, May 2024b.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation, October 2023.

Gaurav Maheshwari, Dmitry Ivanov, and Kevin El Haddad. Efficacy of Synthetic Data as a Benchmark, September 2024.

Mistral. Un Ministral, des Ministraux | Mistral AI. https://mistral.ai/news/ministraux, 2025.

OpenAI. Pricing. https://openai.com/api/pricing/, 2025.

OpenAI, Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, et al. GPT-4o System Card, October 2024.

OpenAI, Ananya Kumar, Jiahui Yu, John Hallman, Michelle Pokrass, Adam Goucher, et al. GPT-4.1 System Card, April 2025a.

OpenAI, Brian Zhang, Eric Mitchell, and Hongyu Ren. OpenAI o3-mini System Card. https://openai.com/index/o3-mini-system-card/, 2025b.

Debalina Ghosh Paul, Hong Zhu, and Ian Bayley. Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review, June 2024.

Weixi Tong and Tianyi Zhang. CodeJudge: Evaluating Code Generation with Large Language Models, October 2024.

Qinyun Wu, Chao Peng, Pengfei Gao, Ruida Hu, Haoyu Gan, Bo Jiang, et al. RepoMasterEval: Evaluating Code Completion via Real-World Repositories, August 2024.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13921–13937, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.859.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, et al. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions, April 2025.

A Further category examples

A.1 API Usage

To illustrate this category, consider the following Python example that evaluates a model's ability to correctly implement asynchronous HTTP requests using the Tornado library:

Example 1: Python API Usage #1

```python
import asyncio
from tornado.httpclient import AsyncHTTPClient

async def fetch_url(url):
    client = AsyncHTTPClient()
    try:
```

Prefix

```python
        response = await client.fetch(url)
```

Golden Completion

```python
    finally:
        client.close()
    return response.body
    ...
```

Suffix

This example demonstrates several key aspects of our API Usage category. First, it tests knowledge of proper asynchronous programming in Python using Tornado's HTTP client. The model must correctly implement the `await` keyword with the appropriate API method (`client.fetch`). The prefix establishes the asynchronous context and resource acquisition, while the suffix ensures proper resource cleanup via the `finally` block. This pattern of establishing context, implementing specialized API calls, and ensuring proper resource management is representative of real-world coding scenarios that developers encounter when working with external libraries.

A.2 Code2NL/NL2Code

The following example demonstrates the Code2NL aspect of this category, where a model must generate appropriate documentation for a C++ financial class:

Example 2: C++ Code2NL/NL2Code #4

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <cassert>
```

Prefix

```cpp
/**
 * @brief Represents a financial instrument in a portfolio.
 * @param name The name of the instrument.
 * @param price The price of the instrument.
 * @param risk The risk factor of the instrument.
 */
```

Golden Completion

```cpp
class FinancialInstrument {
public:
    FinancialInstrument(std::string name, double price, double risk) :
     name_(name), price_(price), risk_(risk) {}

    /// @brief Gets the name of the instrument.
    /// @return The name of the instrument.
    std::string getName() const { return name_; }

    /// @brief Gets the price of the instrument.
    /// @return The price of the instrument.
    double getPrice() const { return price_; }

    /// @brief Gets the risk factor of the instrument.
    /// @return The risk factor of the instrument.
    double getRisk() const { return risk_; }

private:
    std::string name_;
    double price_;
    double risk_;
};

...
```

Suffix

In this example, the model must generate Doxygen-style documentation for a C++ class constructor by inferring the class's purpose and parameters from the subsequent implementation. The prefix contains only standard C++ includes, providing minimal context, while the suffix shows the full class implementation with member functions already documented. The model must understand both the domain context (financial instruments in a portfolio) and the Doxygen documentation format, including the appropriate use of `@brief` for the class description and `@param` tags for each constructor parameter. This tests the model's ability to extract semantic meaning from implementation details and express it in standardized documentation format. The model must "reverse engineer" code understanding into NL explanation, following language-specific documentation conventions that would be expected in professional C++ codebases.

A.3 Low Context

The following example demonstrates how this category tests programming pattern recognition with minimal context in C#:

Example 3: C# Low Context #47

```
using System;
using System.Collections.Generic;
using System.Diagnostics;

public class CustomPaginator<T>
{
    private readonly List<T> _items;
    private readonly int _pageSize;

    public CustomPaginator(List<T> items, int pageSize)
    {
        _items = items;
        _pageSize = pageSize;
    }

    public IEnumerable<IEnumerable<T>> GetPages()
    {
```

Prefix

```
        for (int i = 0; i < _items.Count; i += _pageSize)
        {
            yield return _items.GetRange(i, Math.Min(_pageSize, _items
    .Count - i));
```

Golden Completion

```
        }
    }
}

...
```

Suffix

In this example, the model must implement a pagination iterator with minimal surrounding context. With just the class structure, field definitions, and method signature, the model must infer that an iterator pattern using C#'s `yield return` statement is the idiomatic approach for implementing a paginator. The golden completion demonstrates key C# idioms: using a `for` loop for pagination control, calculating page boundaries with `Math.Min()` to handle the last page case, and most importantly, using the `yield return` pattern to create a lazy enumeration of pages. This pattern is specific to C# and allows for deferred execution of the pagination logic. The model must recognize from the return type `IEnumerable<IEnumerable<T»` that the method should return a sequence of sequences without materializing the entire result set at once. This example highlights how even with minimal context (less than 15 lines total), models must demonstrate deep understanding of language-specific patterns and implement idiomatic solutions that align with the established programming conventions for each language.

A.4 Syntax Completion

The following example demonstrates how this category tests understanding of Java's modern functional syntax features:

Example 4: Java Syntax Completion #5

```java
import java.util.Optional;

public class User {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public String getName() {
        return name;
    }

    ...
}

public class UserService {
    ...

    public String getUserNameOrDefault(String email) {
        Optional<User> userOpt = findUserByEmail(email);
```

Prefix

```java
        return userOpt.map(User::getName).orElse("Unknown User");
```

Golden Completion

```java
    }

    public static void main(String[] args) {
        UserService userService = new UserService();
        String userName = userService.getUserNameOrDefault("
    test@example.com");
        assert userName.equals("Test User") : "Expected 'Test User',
    but got " + userName;
        userName = userService.getUserNameOrDefault("unknown@example.
    com");
        assert userName.equals("Unknown User") : "Expected 'Unknown
    User', but got " + userName;
    }
}
```

Suffix

In this example, the model must demonstrate understanding of Java's `Optional` API and method chaining syntax, which are modern Java features introduced to handle nullable values functionally. The prefix establishes a scenario where a user might be found by email address, returning an `Optional<User>` that could be empty. The golden completion showcases several Java-specific syntax elements in a single line: the functional-style `map` operation with method reference syntax (`User::getName`), followed by chained method invocation with the terminal operation `orElse` to provide a default value. This completion requires precise syntax understanding as it involves multiple Java-specific features: proper method chaining, correct use of method references, and appropriate handling of the `Optional` container. The suffix validates the implementation with assertions testing both the successful case and the default fallback. This example tests the model's ability to produce syntactically correct Java code that leverages modern language features, demonstrating mastery beyond basic language syntax. The conciseness of the golden completion—accomplishing a common nullable-handling pattern in a single expressive line—is representative of idiomatic modern Java programming that models should be capable of generating.

A.5 Pattern Matching

The following example illustrates how this category tests pattern recognition in a Java functional programming context:

Example 5: Java Pattern Matching #29

```java
import java.util.List;
import java.util.ArrayList;
import java.util.function.Function;

// This class demonstrates the use of higher-order functions to apply
    different transformations to a list of integers
public class HigherOrderFunctionsDemo {

    // Method to apply a transformation to a list of integers
    public static List<Integer> transformList(List<Integer> list,
    Function<Integer, Integer> transformation) {
        ...
    }

    // Sample transformations
    public static Function<Integer, Integer> square = x -> x * x;
    public static Function<Integer, Integer> cube = x -> x * x * x;
    public static Function<Integer, Integer> negate = x -> -x;

    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();
        for (int i = 1; i <= 5; i++) {
            numbers.add(i);
        }

        // Apply the square transformation
        List<Integer> squaredNumbers = transformList(numbers, square);
        System.out.println("Squared Numbers: " + squaredNumbers);

        // Apply the cube transformation
        List<Integer> cubedNumbers = transformList(numbers, cube);
        System.out.println("Cubed Numbers: " + cubedNumbers);
```

Prefix

```java
        // Apply the negate transformation
        List<Integer> negatedNumbers = transformList(numbers, negate);
        System.out.println("Negated Numbers: " + negatedNumbers);
```

Golden Completion

```java
        // Assertions
        assert squaredNumbers.equals(List.of(1, 4, 9, 16, 25)) : "
    Squared numbers are incorrect";
        assert cubedNumbers.equals(List.of(1, 8, 27, 64, 125)) : "
    Cubed numbers are incorrect";
        assert negatedNumbers.equals(List.of(-1, -2, -3, -4, -5)) : "
    Negated numbers are incorrect";
    }
}
```

Suffix

In this task, the model extends a functional programming pattern in Java, where higher-order functions are applied to transform a list of integers. The prefix includes two examples (squaring and cubing) using a consistent structure: defining a transformation, applying it via `transformList`, and printing results with a descriptive message. The model must follow this structure and semantics to implement a third transformation (`negate`). This tests the model's ability to recognize and continue idiomatic Java patterns using higher-order functions in a well-defined context.

A.6 Code Purpose Understanding

The following example illustrates and example in the financial domain context:

Example 6: Python Code Purpose Understanding #5

```python
class BankAccount:
    def __init__(self, account_number, balance=0):
        self.account_number = account_number
        self.balance = balance

    ...

    def withdraw(self, amount):
        if amount > 0 and amount <= self.balance:
            self.balance -= amount
            return self.balance
        else:
            raise ValueError("Insufficient funds or invalid withdrawal
     amount")

    def transfer(self, target_account, amount):
```

Prefix

```python
        if amount > 0 and amount <= self.balance:
            self.withdraw(amount)
        else:
            raise ValueError("Insufficient funds or invalid transfer
    amount")
```

Golden Completion

```python
        target_account.deposit(amount)
        return self.balance

...
```

Suffix

In this task, the model implements the `transfer` method in a `BankAccount` class, requiring it to validate that the amount is positive and funds are sufficient. It must reuse the existing `withdraw` method for consistency and raise a domain-specific error if validation fails. This goes beyond syntax, testing the model's ability to reason about object-oriented structure and financial business logic.

B Illustrative model comparison

Example 7: Python Pattern Matching #18

```python
class DataValidator:
    def __init__(self, schema):
        self.schema = schema

    def validate(self, data):
        errors = []
        for field, rules in self.schema.items():
            if field not in data:
                errors.append(f'Missing field: {field}')
            else:
```

```
                value = data[field]
                for rule, expected in rules.items():
                    if rule == 'type' and not isinstance(value,
    expected):
                        errors.append(f'Field {field} should be of
    type {expected.__name__}')
                    elif rule == 'min_length' and len(value) <
    expected:
                        errors.append(f'Field {field} should be at
    least {expected} characters long')
```

Prefix

```
                    elif rule == 'max_length' and len(value) >
    expected:
                        errors.append(f'Field {field} should be at
    most {expected} characters long')
```

Golden Completion

```
                    elif rule == 'max_length' and len(value) >
    expected:
                        errors.append(f'Field {field} should be at
    most {expected} characters long')
                    elif rule == 'required' and not value:
                        errors.append(f'Field {field} is required')
```

Model Completion (DeepSeek-V3)

```
                    elif rule == 'max_length' and len(value) >
    expected:
                        errors.append(f'Field {field} should be at
    most {expected} characters long')
                    elif rule == 'min_value' and value < expected:
                        errors.append(f'Field {field} should be at
    least {expected}')
                    elif rule == 'max_value' and value > expected:
                        errors.append(f'Field {field} should be at
    most {expected}')
```

Model Completion (Claude 3.7 Sonnet)

```
        return errors

schema = {
    'name': {'type': str, 'min_length': 3},
    'age': {'type': int}
}

validator = DataValidator(schema)
data_valid = {'name': 'Alice', 'age': 30}
data_invalid = {'name': 'Al', 'age': 'thirty'}

errors_valid = validator.validate(data_valid)
errors_invalid = validator.validate(data_invalid)

assert errors_valid == []
assert errors_invalid == ['Field name should be at least 3 characters
    long', 'Field age should be of type int']

print('All assertions passed!')
```

Suffix

```python
schema = {
    'name': {'type': str, 'min_length': 3, 'max_length': 10},
    'age': {'type': int}
}

validator = DataValidator(schema)
data_valid = {'name': 'Alice', 'age': 30}
data_invalid_length = {'name': 'Aliceeeeeeee', 'age': 30}
data_invalid_type = {'name': 'Alice', 'age': 'thirty'}

errors_valid = validator.validate(data_valid)
errors_invalid_length = validator.validate(data_invalid_length)
errors_invalid_type = validator.validate(data_invalid_type)

assert errors_valid == []
assert errors_invalid_length == ['Field name should be at most 10
    characters long']
assert errors_invalid_type == ['Field age should be of type int']
```

Assertions

The prefix code presents a `DataValidator` class that implements a validation framework for checking data against a schema. The class has been partially implemented with methods to initialize the validator and validate data, including checks for missing fields, type validation, and minimum length validation. The established pattern is evident in the validation logic structure, where each rule check follows a consistent "if/elif" pattern with appropriate error messages. This demonstrates a real-world scenario where consistent validation rules are essential for maintaining data integrity. The golden completion adds a single rule check for `max_length` that follows the established pattern exactly, validating that field values do not exceed a maximum length and generating an appropriate error message that matches the style of previous validation checks. This completion perfectly extends the pattern established in the prefix and is required to satisfy the assertions. DeepSeek-V3's completion correctly implements the required `max_length` validation rule, matching the golden completion exactly, and then adds only one additional validation rule for the required property. While this additional check is unnecessary for passing the assertions, DeepSeek-V3's completion remains relatively close to the golden standard by limiting its extension to a single additional validation rule that follows the established pattern. Claude 3.7 Sonnet's completion also correctly implements the required `max_length` validation, but then extends the pattern with two unnecessary additional rules for `min_value` and `max_value` validation. These additional rules, while following the established pattern and potentially useful in a real validation system, represent a more significant deviation from the golden completion compared to DeepSeek-V3's response. The inclusion of these two extra validation rules makes Claude 3.7 Sonnet's completion less similar to the golden standard. The suffix code and assertions validate the functionality, confirming that the required `max_length` validation is essential for passing the tests. The example illustrates why DeepSeek-V3 demonstrates stronger Average Cosine Similarity in Pattern Matching compared to Claude 3.7 Sonnet: it more closely adheres to the minimal required pattern extension by adding fewer unnecessary validation rules.

C Detailed LLM-judge experimental results

The heat-map in Figure 4 shows the breakdown of LLM-judge scores by category and languages.

To better understand the relative performance of different models across programming languages, Table 7 presents the LLM-judge scores with 95% confidence intervals for each model-language pair.

D Qualitative examples

We start with an example of a Python model completion that did not successfully execute and did not closely resemble the golden completion from the benchmark.

Example 8: Python Pattern Matching #43

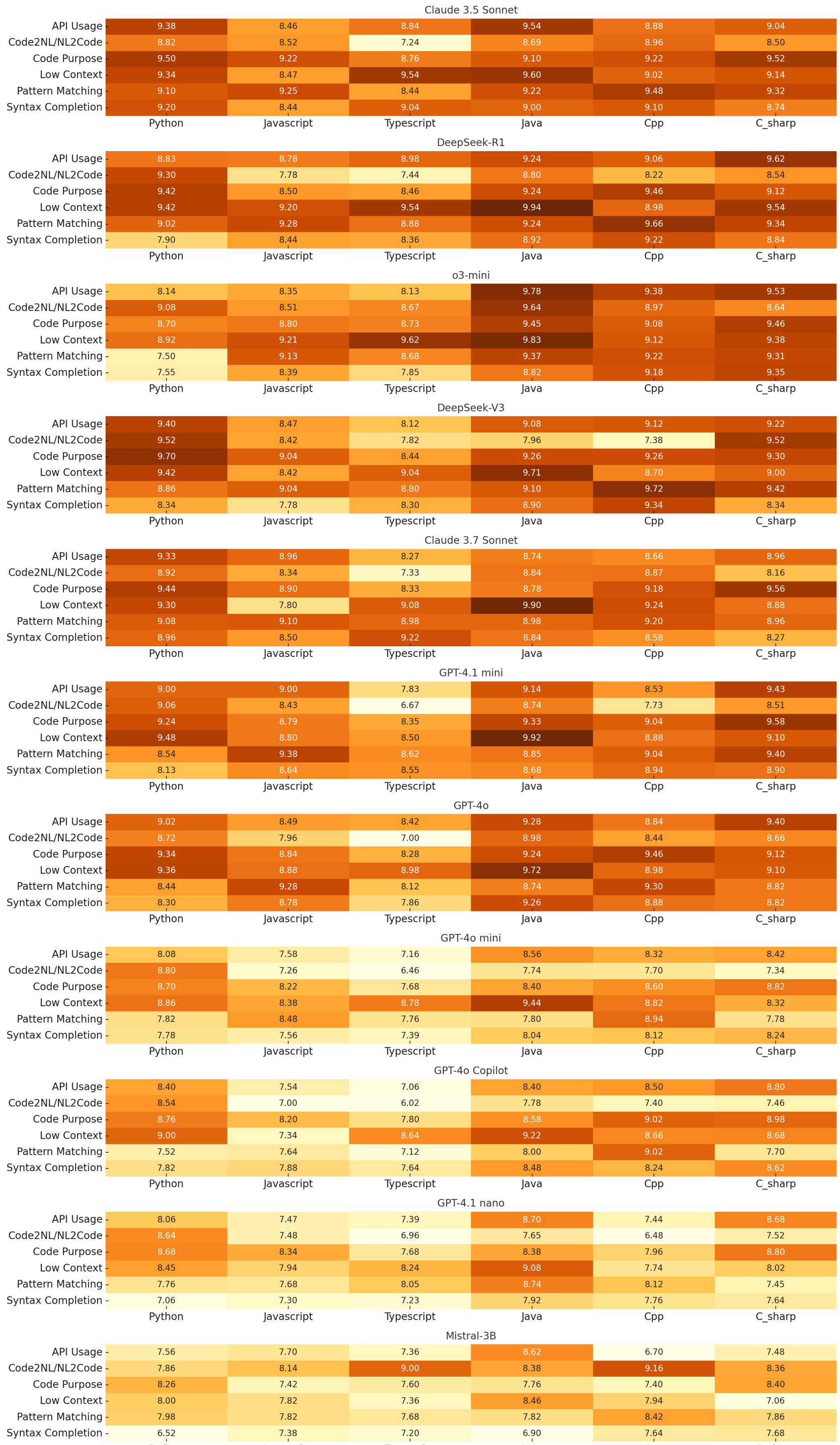

Figure 4: Breakdown of LLM-judge scores across models.

Table 7: Programming language LLM-judge scores of different LLMs with 95% confidence intervals.

Model	C++	C#	Java	JavaScript	Python	TypeScript
Claude 3.7 Sonnet	8.96 (8.71-9.19)	8.80 (8.51-9.07)	9.01 (8.75-9.25)	8.60 (8.28-8.90)	9.18 (8.96-9.39)	8.53 (8.23-8.83)
Claude 3.5 Sonnet	9.11 (8.89-9.33)	9.04 (8.81-9.27)	9.19 (8.96-9.42)	8.73 (8.44-8.99)	9.24 (9.03-9.43)	8.64 (8.35-8.92)
GPT-4o	8.98 (8.73-9.23)	8.99 (8.74-9.22)	9.20 (8.99-9.41)	8.71 (8.44-8.95)	8.90 (8.65-9.14)	8.11 (7.79-8.41)
DeepSeek-V3	8.92 (8.64-9.18)	9.13 (8.89-9.36)	9.00 (8.73-9.25)	8.53 (8.22-8.82)	9.16 (8.94-9.36)	8.42 (8.11-8.70)
GPT-4.1 mini	8.71 (8.42-8.99)	9.16 (8.93-9.37)	9.12 (8.87-9.35)	8.84 (8.58-9.08)	8.96 (8.72-9.20)	8.09 (7.75-8.43)
o3-mini	9.17 (8.92-9.39)	9.29 (9.06-9.50)	9.48 (9.30-9.65)	8.73 (8.43-9.01)	8.43 (8.08-8.73)	8.60 (8.29-8.90)
GPT-4o Copilot	8.47 (8.15-8.78)	8.37 (8.07-8.67)	8.41 (8.07-8.73)	7.60 (7.24-7.95)	8.37 (8.06-8.68)	7.38 (7.03-7.73)
GPT-4o mini	8.42 (8.10-8.72)	8.15 (7.83-8.47)	8.33 (7.98-8.65)	7.91 (7.60-8.22)	8.36 (8.06-8.66)	7.54 (7.21-7.87)
DeepSeek-R1	9.10 (8.87-9.32)	9.17 (8.95-9.36)	9.23 (9.01-9.43)	8.66 (8.38-8.95)	8.98 (8.73-9.20)	8.61 (8.32-8.88)
GPT-4.1 nano	7.58 (7.21-7.94)	8.02 (7.67-8.35)	8.41 (8.10-8.71)	7.71 (7.37-8.05)	8.15 (7.82-8.46)	7.59 (7.24-7.92)
Ministral-3B	7.88 (7.56-8.19)	7.81 (7.45-8.15)	7.99 (7.64-8.32)	7.71 (7.36-8.05)	7.74 (7.41-8.05)	7.70 (7.35-8.03)

```
import matplotlib.pyplot as plt
import numpy as np

class AnalyticsReport:
    def __init__(self, data):
        self.data = data

    def generate_histogram(self, column):
        plt.hist(self.data[column], bins=10, alpha=0.75)
        plt.title(f'Histogram of {column}')
        plt.xlabel(column)
        plt.ylabel('Frequency')
        plt.grid(True)

    def generate_line_chart(self, x_column, y_column):
        plt.plot(self.data[x_column], self.data[y_column], marker='o',
    linestyle='-')
        plt.title(f'Line Chart of {y_column} vs {x_column}')
        plt.xlabel(x_column)
        plt.ylabel(y_column)
        plt.grid(True)
```

Prefix

```
    def generate_bar_chart(self, x_column, y_column):
        plt.bar(self.data[x_column], self.data[y_column], alpha=0.75)
        plt.title(f'Bar Chart of {y_column} vs {x_column}')
        plt.xlabel(x_column)
        plt.ylabel(y_column)
        plt.grid(True)
```

Golden Completion

```
    def generate_scatter_plot(self, x_column, y_column):
        plt.scatter(self.data[x_column], self.data[y_column], alpha
   =0.75)
        plt.title(f'Scatter Plot of {y_column} vs {x_column}')
        plt.xlabel(x_column)
        plt.ylabel(y_column)
        plt.grid(True)
```

Model Completion (Claude 3.7 Sonnet)

```
    def generate_scatter_plot(self, x_column, y_column):
        plt.scatter(self.data[x_column], self.data[y_column], alpha
   =0.75)
        plt.title(f'Scatter Plot of {y_column} vs {x_column}')
        plt.xlabel(x_column)
        plt.ylabel(y_column)
        plt.grid(True)
```

```python
# Example usage
if __name__ == '__main__':
    data = {
        'age': np.random.randint(20, 50, size=100),
        'height': np.random.randint(150, 200, size=100),
        'weight': np.random.randint(50, 100, size=100)
    }
    report = AnalyticsReport(data)
    report.generate_histogram('age')
    report.generate_line_chart('height', 'weight')
    plt.show()
```

Suffix

```python
data = {
    'age': np.random.randint(20, 50, size=100),
    'height': np.random.randint(150, 200, size=100),
    'weight': np.random.randint(50, 100, size=100)
}
report = AnalyticsReport(data)
report.generate_bar_chart('age', 'weight')
assert hasattr(report, 'generate_bar_chart')
assert callable(getattr(report, 'generate_bar_chart'))
assert plt.gcf().get_axes()[-1].get_title() == 'Bar Chart of weight vs
    age'
plt.close('all')
```

Assertions

The prefix code shows an AnalyticsReport class with two visualization methods: a histogram and a line chart generator. Both methods follow a consistent pattern of creating a plot, adding a title, labels, and grid. The golden completion provides a `generate_bar_chart` method that follows the same pattern established in the prefix code. This implementation is crucial for the assertions that test functionality later in the code. The golden completion correctly maintains the structural pattern of the other visualization methods, ensuring consistency in the codebase. Claude 3.7 Sonnet, however, generated a `generate_scatter_plot` method, which already exists in the suffix, instead of the expected bar chart. This implementation not only deviates from the task of creating a bar chart, but also fails to pass the assertions that specifically test for the presence and functionality of a `generate_bar_chart` method. While the scatter plot follows the general pattern of the other visualization methods, it represents an incorrect solution to the task. The suffix code shows that the expected implementation should have been a bar chart function, not a scatter plot. This discrepancy reveals the model's failure to properly understand the context and requirements of the task.

Now, we proceed with an example of a Python model completion that successfully executed but did not closely resemble the golden completion from the benchmark.

Example 9: Python Code2NL/NL2Code #15

```python
import requests

def fetch_data_from_api(url):
    """
    Fetches data from the given API URL.

    Parameters:
    url (str): The API endpoint to fetch data from.

    Returns:
    dict: The JSON response from the API parsed into a dictionary.

    Raises:
    ValueError: If the response status code is not 200.
```

```python
    """
    response = requests.get(url)
    if response.status_code != 200:
        raise ValueError("API request failed with status code {}".
    format(response.status_code))
    return response.json()

# Example usage:
# data = fetch_data_from_api('https://api.example.com/data')
# print(data)
```

Prefix

```python
# Call the fetch_data_from_api function with a sample URL and handle
    the response
```

Golden Completion

```python
def main():
    try:
        data = fetch_data_from_api('https://jsonplaceholder.typicode.
    com/todos/1')
        assert isinstance(data, dict), 'The fetched data should be a
    dictionary'
        print("Data fetched successfully.")
    except ValueError as ve:
        print("Error fetching data:", ve)

    # Additional logic can be placed here

if __name__ == "__main__":
    main()
```

Model Completion (Claude 3.7 Sonnet)

```python
def main():
    try:
        data = fetch_data_from_api('https://jsonplaceholder.typicode.
    com/todos/1')
        assert isinstance(data, dict), 'The fetched data should be a
    dictionary'
        print("Data fetched successfully.")
    except ValueError as ve:
        print("Error fetching data:", ve)

    # Additional logic can be placed here

if __name__ == "__main__":
    main()
```

Suffix

```python
assert 'fetches data from the given api url' in fetch_data_from_api.
    __doc__.lower()
assert 'parameters:' in fetch_data_from_api.__doc__.lower()
assert 'returns:' in fetch_data_from_api.__doc__.lower()
assert 'raises:' in fetch_data_from_api.__doc__.lower()
assert 'url (str)' in fetch_data_from_api.__doc__.lower()
assert 'dict: the json response from the api' in fetch_data_from_api.
    __doc__.lower()
assert 'valueerror: if the response status code is not 200' in
    fetch_data_from_api.__doc__.lower()
try:
    data = fetch_data_from_api('https://jsonplaceholder.typicode.com/
    nonexistent')
```

```python
    assert isinstance(data, dict), 'The fetched data should be a
    dictionary'
except ValueError as ve:
    assert str(ve) == 'API request failed with status code 404'
```

Assertions

The prefix code presents a `fetch_data_from_api` function that retrieves data from a given API URL, with comprehensive documentation including parameters, return values, and potential exceptions. The code also includes commented examples of how to use the function, suggesting a pattern for implementation. The golden completion adds a clear comment that describes what follows: a call to the function with error handling. This simple, descriptive comment serves as NL guidance for the subsequent code block, following good documentation practices. The comment effectively bridges the gap between the function definition and its implementation, making the code more readable and maintainable. This approach aligns with common user behavior observed in telemetry data. Claude 3.7 Sonnet, however, completely duplicated the `main()` function that already exists in the suffix code. While this implementation technically works and passes the assertions, it demonstrates a failure to recognize that the completion should complement rather than duplicate the suffix code. The model has essentially copied the exact content of the suffix rather than providing a meaningful transition between the prefix and suffix.

Now, we proceed with an example of a Python model completion that did not successfully execute but closely resembles the golden completion from the benchmark.

Example 10: Python Pattern Matching #4

```python
class DataValidator:
    def __init__(self, schema):
        self.schema = schema

    def validate(self, data):
        """Validate data against the schema."""
        errors = []
        for field, field_type in self.schema.items():
            if field not in data:
                errors.append(f"Missing field: {field}")
            elif not isinstance(data[field], field_type):
                errors.append(f"Incorrect type for field: {field}")
```

Prefix

```python
            else:
                if isinstance(data[field], str) and not data[field].
    strip():
                    errors.append(f"Empty field: {field}")
```

Golden Completion

```python
            elif not isinstance(data[field], field_type):
                errors.append(f"Incorrect type for field: {field}")
```

Model Completion (DeepSeek-V3)

```python
        return errors

schema = {
    "name": str,
    "age": int,
    "email": str
}

validator = DataValidator(schema)
data = {
```

```
    "name": "John Doe",
    "age": 30,
    "email": "john.doe@example.com"
}

errors = validator.validate(data)
assert errors == [], f"Unexpected errors: {errors}"

invalid_data = {
    "name": "John Doe",
    "age": "thirty",
    "email": "john.doe@example.com"
}

errors = validator.validate(invalid_data)
assert "Incorrect type for field: age" in errors, f"Expected type
    error for age, got: {errors}"
assert len(errors) == 1, f"Expected one error, got: {len(errors)}"

invalid_data = {
    "age": 30,
    "email": "john.doe@example.com"
}

errors = validator.validate(invalid_data)
assert "Missing field: name" in errors, f"Expected missing field error
     for name, got: {errors}"
assert len(errors) == 1, f"Expected one error, got: {len(errors)}"
```

Suffix

```
schema = {
    "name": str,
    "age": int,
    "email": str
}

validator = DataValidator(schema)
data = {
    "name": "",
    "age": 30,
    "email": "john.doe@example.com"
}

errors = validator.validate(data)
assert "Empty field: name" in errors, f"Expected empty field error for
     name, got: {errors}"
assert len(errors) == 1, f"Expected one error, got: {len(errors)}"
```

Assertions

The prefix code presents a `DataValidator` class that checks data against a schema, validating both field presence and type matching. The validation method accumulates errors in a list and handles two specific validation cases: missing fields and incorrect data types. The golden completion extends the validation logic by adding a third check specifically for string fields, ensuring they aren't empty after stripping whitespace. The golden completion correctly introduces this check as an else branch after the type validation, maintaining the logical flow of the validation process. DeepSeek-V3, however, duplicated the existing type validation check rather than adding the new empty string validation logic. This duplication creates a logical error, as the same condition (`elif not isinstance(data[field], field_type)`) appears twice in sequence. While DeepSeek-V3's completion structurally resembles the golden completion in that it maintains the pattern of adding conditions related to `data[field]` with appropriate error messages, it fails to introduce the new validation logic needed to pass the assertions in the test suite. The assertion tests specifically verify the ability to detect empty string fields, which the model's completion does not implement. This example demonstrates how a model's completion can closely resemble the golden solution in structure while still containing critical logical errors that prevent proper execution.

E Additional benchmark and experimental details

E.1 Python execution

Our Python evaluation instance evaluation methodology implements a robust, secure, and reproducible execution environment. Each evaluation instance consists of four components: a context prefix, a golden completion (or model-generated completion during evaluation), a context suffix, and assertion statements that verify correctness. The execution pipeline first combines these components into a complete Python program with additional safeguards. We automatically insert a `matplotlib` non-interactive backend configuration to prevent `plt.show()` calls from blocking execution, and we handle environment variables securely to provide necessary API access while maintaining isolation. Test execution occurs in a controlled subprocess with a 30-second timeout to prevent infinite loops, and we implement comprehensive error handling for execution failures. When dependency-related errors occur, our system automatically attempts to install the missing packages using `pip` before retrying execution. Each evaluation instance runs in its own isolated environment to prevent cross-contamination between tests, with proper cleanup of temporary files after execution. This approach allows us to comprehensively evaluate models on executable code with real-world dependencies, providing a high-fidelity measure of code completion performance that aligns with actual developer workflows. The entire evaluation pipeline includes detailed logging and reporting functionality, generating both human-readable reports and structured JSON output for further analysis.

E.2 Benchmark generation prompts

To create **DevBench**'s diverse and realistic evaluation instances, we developed specialized generation prompts that captured the nuances of each programming language and code completion category. These structured prompts guided the GPT-4o model to create evaluation instances that accurately reflect real-world coding scenarios identified in our telemetry analysis. Each prompt was meticulously crafted with specific instructions detailing the characteristic patterns, expected structures, and quality requirements for generating valid evaluation instances. The prompts ensured consistent formatting while maintaining language-specific idioms and patterns, balancing standardization with authentic coding styles. In this section, we present the template prompts used for each language-category pair, demonstrating how we systematically encoded the insights from our telemetry analysis into generative instructions that produced high-quality synthetic evaluation instances while maintaining evaluation instance realism. Due to space constraints, we only include one C++ prompt here; the complete collection of prompts for all languages and categories is available in our code repository.

C++: API Usage Prompts

```
API_USAGE_SYSTEM_PROMPT = """
You are an expert C++ developer tasked with creating benchmark
    examples for testing rare API usage and uncommon library function
    capabilities in large language models.
Your role is to generate high-quality, realistic coding scenarios that
     effectively test an LLM's ability to recognize and continue
    established patterns in code involving uncommon APIs and library
    functions.

Your output should be a single JSON object formatted as a JSONL entry.
     The code must be fully executable C++ that passes all assertions.

Key Responsibilities:
1. Generate diverse examples from these API categories (rotate through
     them, don't focus only on file operations or network protocols):
     - Text and font processing (HarfBuzz, FreeType, ICU)
     - Graphics and math libraries (DirectXMath, Eigen, GLM, OpenGL)
     - Security/cryptography APIs (OpenSSL, Botan, Crypto++, wolfSSL)
```

```
    - System-level APIs (Windows SDK, POSIX, Linux Kernel, BSD, Mach)
    - Standard libraries (C Standard Library, C++ Standard Library,
   GNU C Library)
    - Web API integration (libcurl, Boost.Beast, cpp-httplib,
   cpprestsdk)
    - Machine learning libraries (OpenCV, TensorFlow C++, PyTorch C++,
     ONNX)
    - Cloud services (AWS SDK for C++, Azure SDK for C++, gRPC)
    - Database interfaces (SQLite, MySQL Connector C++, MongoDB C++
   Driver, Redis)
    - File formats and parsing (RapidJSON, nlohmann/json, tinyxml2,
   yaml-cpp)
    - Web frameworks (Drogon, Crow, oatpp, Pistache)
    - Network protocols (Boost.Asio, ZeroMQ, nanomsg)
    - Scientific computing (Eigen, Armadillo, Intel MKL, BLAS, LAPACK)
    - GUI frameworks (Qt, wxWidgets, ImGui, GTK, FLTK)
    - Multimedia (SDL, FFmpeg, OpenAL, libsndfile)
    - Compression (zlib, bzip2, LZMA, LZ4, Zstandard)
    - Cross-platform development (Boost, Qt, wxWidgets)
    - Mobile development (Android NDK, iOS SDK, Core Foundation)
    - Testing frameworks (Google Test, Catch2, Boost.Test)
    - Hardware acceleration (Intel Intrinsics, ARM NEON, CUDA, OpenCL)
    - Legacy/deprecated APIs

2. Ensure patterns are clear and identifiable even with uncommon or
   deprecated APIs
3. Create ground truth completions that represent best practices while
     handling API versioning
4. Write assertions that meaningfully test both API correctness and
   parameter ordering
5. Provide clear justification for why the example makes a good test
   case
6. Ensure code quality:
    - All code must be fully executable C++
    - All assertions must pass when code is run
    - Include necessary includes and namespaces
    - Handle cleanup of resources
    - Use proper exception handling
    - Include minimal working examples
    - Mock external dependencies where needed
7. Write robust assertions that:
    - Verify actual API behavior
    - Test parameter ordering
    - Check error conditions
    - Validate return values
    - Mock external resources

When generating examples:
1. Focus on less common library functions and domain-specific APIs
2. Test the model's handling of deprecated but valid API patterns
3. Ensure patterns include correct parameter ordering and naming
   conventions
4. Include edge cases in API usage where relevant
5. Keep code focused on demonstrating rare but valid API interactions
"""

API_USAGE_USER_PROMPT = """
You are helping create a benchmark for rare API usage capabilities.
   Your task is to generate a coding scenario that tests an LLM's
   ability to recognize and
complete patterns in C++ code involving uncommon or deprecated APIs.

Generate a single JSONL entry testing rare API usage capabilities.
   Choose from one of these categories (rotate through them, don't
   focus only on file operations or network protocols):
```

```
    - Text and font processing (HarfBuzz, FreeType, ICU)
    - Graphics and math libraries (DirectXMath, Eigen, GLM, OpenGL)
    - Security/cryptography APIs (OpenSSL, Botan, Crypto++, wolfSSL)
    - System-level APIs (Windows SDK, POSIX, Linux Kernel, BSD, Mach)
    - Standard libraries (C Standard Library, C++ Standard Library,
   GNU C Library)
    - Web API integration (libcurl, Boost.Beast, cpp-httplib,
   cpprestsdk)
    - Machine learning libraries (OpenCV, TensorFlow C++, PyTorch C++,
    ONNX)
    - Cloud services (AWS SDK for C++, Azure SDK for C++, gRPC)
    - Database interfaces (SQLite, MySQL Connector C++, MongoDB C++
   Driver, Redis)
    - File formats and parsing (RapidJSON, nlohmann/json, tinyxml2,
   yaml-cpp)
    - Web frameworks (Drogon, Crow, oatpp, Pistache)
    - Network protocols (Boost.Asio, ZeroMQ, nanomsg)
    - Scientific computing (Eigen, Armadillo, Intel MKL, BLAS, LAPACK)
    - GUI frameworks (Qt, wxWidgets, ImGui, GTK, FLTK)
    - Multimedia (SDL, FFmpeg, OpenAL, libsndfile)
    - Compression (zlib, bzip2, LZMA, LZ4, Zstandard)
    - Cross-platform development (Boost, Qt, wxWidgets)
    - Mobile development (Android NDK, iOS SDK, Core Foundation)
    - Testing frameworks (Google Test, Catch2, Boost.Test)
    - Hardware acceleration (Intel Intrinsics, ARM NEON, CUDA, OpenCL)
    - Legacy/deprecated APIs

CRITICAL JSON FORMATTING REQUIREMENTS:
1. Your response MUST be a syntactically valid JSON object
2. PROPERLY ESCAPE all special characters in strings:
   - Use \\" for double quotes inside strings
   - Use \\n for newlines
   - Use \\t for tabs
   - Use \\\\ for backslashes
3. The entire JSON object must be on a SINGLE LINE
4. Do NOT include formatting or indentation outside the JSON structure
5. DO NOT use markdown code blocks (```) in your response
6. Test your JSON structure before completing your response

Required JSON fields:
- id: A unique numeric identifier
- testsource: Use "synthbench-api-usage"
- language: "cpp"
- prefix: The code that comes before the completion (may or may not
    establish the API pattern)
- suffix: The code that follows the completion (may or may not
    establish the API pattern) - should be DIFFERENT from the golden
    completion AND should include necessary assertions
- golden_completion: The correct API implementation that maintains
    consistency with prefix/suffix and will pass all assertions
- LLM_justification: Explain why this is a good test case and the
    context behind it
- assertions: Leave this field as an empty string - all assertions
    should be integrated into the suffix code

CRITICAL JSON FIELD REQUIREMENTS:
1. ALWAYS include ALL required JSON fields listed above, even if empty
2. The "assertions" field MUST be present with an empty string value:
    "assertions": ""
3. Do NOT omit any fields from your JSON object
4. Format example showing required empty assertions field:
   {"id": "42", ..., "assertions": ""}
5. INCORRECT: {"id": "42", ...} - missing assertions field

CRITICAL CHANGE - NEW SUFFIX REQUIREMENTS:
```

```
1. The suffix must contain both execution code AND assertion code
2. Include assert() statements DIRECTLY IN THE SUFFIX at the
    appropriate places
3. All assertions must be placed in the same function/class as the
    code being tested
4. DO NOT create separate assertion functions or classes
5. Place assertions immediately after the code that should be tested
6. Never duplicate any golden_completion code in the suffix
7. The assertions must pass when the combined prefix +
    golden_completion + suffix is run

Critical Requirements for Avoiding Duplication:
1. The golden_completion field should ONLY contain the solution code
    that fills in the gap
2. The suffix must contain DIFFERENT code that follows after the
    completion
3. Do NOT repeat any golden_completion code in the suffix
4. The suffix field should NEVER duplicate the golden_completion code
5. There should be a clear DISTINCTION between what goes in
    golden_completion vs suffix
6. Ensure clear SEPARATION between completion and suffix content

Include Requirements:
1. Do NOT include headers unless they are ACTUALLY USED in at least
    one of:
   - prefix
   - suffix (including assertions)
   - golden_completion
2. Every included header must serve a clear purpose
3. Do not include "just in case" headers that aren't used
4. All required includes must appear in the prefix section
5. If an include is only needed for the golden_completion, it must
    still appear in the prefix
6. Make sure to include <cassert> header for assert() statements

PREFIX LENGTH REQUIREMENTS - CRITICAL:
1. The PREFIX section MUST be SUBSTANTIALLY LONGER than other sections
2. The prefix MUST be AT LEAST 50-60 lines of code - this is an
    absolute requirement
3. Provide extensive context and setup code in the prefix
4. Include helper functions, utility classes, and related code
    structures
5. Add detailed comments and explanations within the prefix
6. The prefix should demonstrate a comprehensive but incomplete
    implementation
7. Add relevant constants, configuration objects, and data structure
    initialization

Indentation requirements:
1. All code sections must maintain consistent indentation
2. If code is inside a function/class:
- The prefix should establish the correct indentation level
- The golden_completion must match the prefix's indentation
- The suffix must maintain the same indentation context
- Assertions should be at the appropriate scope level
3. Ensure proper dedenting when exiting blocks
4. All code blocks must be properly closed

The API pattern can be established either in the prefix or suffix code
    .
The golden completion should demonstrate understanding and correct
    usage of the API pattern regardless of where it is established.

Code requirements:
1. Must be fully executable C++ code
```

```
2. All assertions must pass when run
3. Include all necessary headers and namespaces
4. Mock external dependencies
5. Clean up resources properly
6. Handle errors appropriately
7. Assertions must be placed BEFORE cleanup code
8. Resource cleanup must be in the suffix AFTER all assertions
9. All assertions must complete before any cleanup occurs

CRITICAL CODE STRUCTURE REQUIREMENTS:
1. NEVER place code outside of functions or classes
2. ALL code must be contained within proper C++ scope boundaries
3. DO NOT place assertions or standalone code statements at the global
   /namespace level
4. ALL assertions must be contained within functions (such as main()
   or other functions)
5. ALWAYS ensure code is properly nested within appropriate class and
   function structures
6. NEVER generate code that would compile as a partial class
7. NEVER duplicate class definitions - each class must be defined only
    once
8. Verify that the beginning and end of classes and functions are
   properly matched with braces {}
9. DO NOT leave any code statements outside of function bodies
10. Place all assertions within appropriate functions (main(), test(),
    etc.)

CRITICAL ASSERTION PLACEMENT:
1. All assert() statements must be placed DIRECTLY IN THE SUFFIX code
2. Assertions should be placed immediately after the code that needs
   to be verified
3. Assertions must be within the same function as the code being
   tested
4. Assertions must be executed BEFORE any cleanup code
5. Assertions must be properly indented to match the surrounding code
   structure
6. Use assert(condition) format for all assertions
7. Make sure <cassert> is included for assert() statements

Requirements:
1. The scenario should demonstrate a clear pattern recognizable with
   the given context
2. The completion section should focus on rare library functions
3. The pattern should follow correct API conventions across different
   versions
4. Ground truth should demonstrate proper parameter ordering
5. Assertions should verify API behavior and parameter correctness
6. Include comments indicating API version compatibility and parameter
    requirements

Format your response as a single line JSON object with newlines
   escaped appropriately.

Example format:
{"id": "1", "testsource": "synthbench-api-usage", "language": "cpp", "
   prefix": "...", "suffix": "...", "golden_completion": "...", "
   LLM_justification": "...", "assertions": "..."}

VALIDATION CHECKLIST BEFORE SUBMITTING:
1. Have you properly escaped ALL special characters?
2. Is your entire response a single, valid JSON object?
3. Are all string values properly quoted and terminated?
4. Have you verified there are no unescaped newlines in your strings?
5. Have you checked for balanced quotes and braces?
6. Is your prefix at least 50-60 lines of code?
```

```
7. Have you used clear distinctions between golden_completion and
   suffix?
8. Have you included all assertions DIRECTLY IN THE SUFFIX code?
9. Have you verified that assertions will pass when the code is
   executed?
10. Is the assertions field included with an empty string value ("
   assertions": "")?
11. Have you verified that ALL required fields are present in your
   JSON?
12. Have you verified your example is NOT one of the prohibited
   trivial examples?
13. Does your example meet ALL the complexity validation criteria?
14. Does your example demonstrate genuinely advanced C++ features?

Important:
- Never place cleanup code before assertions
- Keep all verification code before any cleanup
- Ensure resources exist when assertions run
- Use proper try/finally blocks if needed
- Maintain correct execution order
- ALL ASSERTIONS SHOULD BE IN THE SUFFIX, not in a separate assertions
    field

Ensure the example is self-contained and can be evaluated
   independently. All assertions must pass when run.
Use proper escaping for newlines/quotes and maintain indentation in
   the escaped strings.

"""
```

E.3 Evaluation prompt

For our model evaluation process, we implemented a carefully designed prompt template focused on precise code completion tasks. After initial experimentation revealed that different prompt formats could significantly impact model performance, including Claude 3.7 Sonnet, due to formatting issues, we selected a structured instruction-based approach that addresses common failure modes. Our code repository contains the full evaluation prompt.

The selected prompt format provides clear examples demonstrating proper replacement behavior in various scenarios, explicitly instructing models to maintain correct indentation and avoid duplicating existing code structures. By standardizing the input format with clear `#TODO: You Code Here markers` and providing explicit instructions against common mistakes, we created a more level evaluation environment that better isolates models' code understanding capabilities from prompt interpretation abilities.

This evaluation prompt design aligns with real-world code completion scenarios where maintaining contextual formatting is essential for functional correctness, ensuring our benchmark more accurately reflects models' practical utility in development environments. Performance differences observed between models using this standardized prompt more reliably indicate their intrinsic code completion capabilities rather than their ability to navigate ambiguous or unstructured prompting patterns.

E.4 Full similarity metrics by category

While the main paper presented similarity metrics for four representative foundation models to maintain clarity and focus on key trends, we provide the complete results across all evaluated models in Table 8. This expanded view offers a more comprehensive comparison of model performance across different categories and similarity dimensions.

F Limitations and future directions

While **DevBench** represents a significant advancement in code generation evaluation, we identify several opportunities for future enhancement and extension.

Table 8: Similarity metrics across task categories. All reported results used a temperature of 0.2.

Model	Average Cosine Similarity						Line 0 Exact Match Rate (%)					
	API Usage	Code2NL NL2Code	Purpose Underst.	Low Context	Pattern Matching	Syntax Compl.	API Usage	Code2NL NL2Code	Purpose Underst.	Low Context	Pattern Matching	Syntax Compl.
Claude 3.7 Sonnet	0.51	0.58	0.65	0.75	0.69	0.61	30.0	49.67	49.0	59.33	50.0	42.67
Claude 3.5 Sonnet	0.60	0.56	0.73	0.77	0.73	0.67	35.67	48.33	54.33	58.33	54.0	47.33
GPT-4o	0.68	0.50	0.72	0.82	0.71	0.65	43.33	43.67	56.67	65.0	55.0	45.33
DeepSeek-V3	0.62	0.53	0.73	0.79	0.74	0.67	37.0	45.0	54.33	59.0	56.67	45.33
GPT-4.1 mini	0.62	0.50	0.72	0.78	0.67	0.62	37.33	42.0	54.67	61.67	50.67	44.33
o3-mini	0.58	0.43	0.67	0.74	0.55	0.56	34.33	34.67	47.33	55.67	38.0	34.33
GPT-4o Copilot	0.61	0.52	0.71	0.77	0.66	0.63	40.67	44.67	54.33	56.67	50.33	42.33
GPT-4o mini	0.62	0.46	0.70	0.78	0.64	0.59	37.67	36.67	51.0	57.0	46.67	38.0
DeepSeek-R1	0.63	0.47	0.70	0.77	0.68	0.64	35.0	38.0	50.67	54.0	48.67	42.67
GPT-4.1 nano	0.57	0.39	0.64	0.73	0.58	0.54	30.0	29.0	42.33	49.0	38.33	32.0
Ministral-3B	0.45	0.32	0.51	0.50	0.46	0.43	18.67	22.0	28.0	26.33	23.67	19.33

F.1 Expanding benchmark generation diversity

Our synthetic, telemetry-driven generation approach effectively prevents data contamination and limits bias by leveraging GPT-4o as the generation model. To further enhance diversity, future iterations could incorporate multiple foundation models with varied training backgrounds. This approach will maintain our telemetry-driven, human-validated methodology while expanding stylistic diversity.

The derivation of test categories from real-world telemetry data grounds our benchmark in authentic developer experiences. Building on this foundation, future research could explore federated learning approaches that enable even closer alignment with real developer interactions while maintaining privacy safeguards.

F.2 Enhancing evaluation frameworks

The complementary evaluation metrics we employ (Pass@1, similarity-based metrics, and LLM-judge assessments) provide multidimensional insights into model performance. The occasional divergence between these metrics—such as cases where higher syntactic similarity does not correlate with functional correctness—highlights an opportunity to develop composite metrics that better capture the full spectrum of code quality dimensions relevant to developers.

Our LLM-judge uses o3-mini as the scoring model, selected for its favorable bias profile as documented in the OpenAI System Card showing lowest bias on discrimination tasks [OpenAI et al., 2025b]. Future work could explore ensemble judging approaches, human-in-the-loop calibration, or contrastive evaluation techniques that specifically control for stylistic biases, allowing for even more robust evaluation.

Execution-based functional correctness evaluation is currently fully implemented only for Python (300 instances). For the remaining five languages, we have implemented multiple safeguards to ensure golden completion reliability: automatic syntax validation using language-specific compilers and parsers for all test cases, expert review process where experienced developers verify that all golden completions satisfy embedded assertions, spot-check execution testing performed manually across samples from all five non-Python languages, and prompting safeguards that enforce syntactically correct outputs with assertions, achieving 100% correctness in Python and high reliability in validated non-Python samples.

Extending execution-based evaluation to all languages is our highest priority. We are developing containerized execution environments for non-Python languages, leveraging language-specific testing frameworks and dependency management systems. This infrastructure development is already underway.

F.3 Broadening coverage scope

DevBench currently provides strong coverage of code completion scenarios while offering opportunities to expand into additional development activities. Future extensions could apply our methodology to generate synthetic evaluation instances for code refactoring, debugging, multi-file architecture design, and system-level programming challenges—further enriching the evaluation landscape.

Our language coverage, which already includes six major programming languages (Python, JavaScript, TypeScript, Java, C++, and C#), provides a foundation for expansion. Future iterations could incorporate emerging languages such as Rust, Go, Ruby, and Swift, as well as develop more complex, multi-stage evaluation instances that reflect the challenges of professional software engineering.

F.4 Optimizing resource efficiency

The benchmark generation process, while relatively affordable using current API pricing (approximately $5.00/1M input tokens and $20.00/1M output tokens for GPT-4o [OpenAI, 2025]), presents opportunities for further efficiency improvements. Future work could provide streamlined tools and templates for benchmark extension, reducing the expertise required to create custom evaluation instances while maintaining quality standards.

Response latency represents another dimension deserving further exploration, as it can impact developer workflow and productivity. Incorporating systematic latency evaluation alongside quality metrics would provide a more holistic view of the practical trade-offs involved in model selection.

F.5 Advancing fairness and inclusivity

The telemetry data that informs our benchmark categories derives from diverse developer interactions, offering an opportunity to explicitly analyze potential implicit biases in programming styles, paradigms, or practices. Future research could conduct systematic analyses of representation across different programming communities and traditions, ensuring the benchmark remains equitable and inclusive.

Performance disparities across programming languages present another avenue for methodological refinement. Future extensions could develop language-specific normalization techniques or targeted improvements for underrepresented languages, ensuring fairness across diverse developer communities and technical ecosystems.

G Broader impacts

Our **DevBench** benchmark has several potential positive societal impacts. By enabling more accurate evaluation of code completion models, our work can lead to improved developer productivity tools that reduce repetitive coding tasks, decrease the time required to implement software solutions, and potentially lower barriers to entry in programming by assisting novice developers. More accurate code completion could also improve software quality by suggesting well-tested patterns and reducing common programming errors, potentially leading to more reliable and secure software systems.

However, we also acknowledge several potential negative impacts. First, there are fairness considerations related to programming language representation; our benchmark's coverage of six languages, while broader than many existing benchmarks, still represents a limited subset of the programming ecosystem. This may lead to uneven improvements across programming languages, potentially disadvantaging developers who work primarily with languages not included in our benchmark. Second, there are potential job market implications if increasingly capable code completion systems begin to automate significant portions of software development tasks, potentially affecting employment opportunities for certain types of programming roles.

Additionally, we recognize that improvements in code generation capabilities could have security implications. While our benchmark focuses on code completion rather than full program generation, advances in code synthesis could potentially be misused to generate malicious code more efficiently or to exploit vulnerabilities in existing systems. To mitigate these concerns, we have designed our benchmark to emphasize proper API usage, security patterns, and code quality metrics rather than merely measuring functional correctness.

To address these concerns, we have made our benchmark and methodology publicly available to enable community scrutiny, external validation, and continuous improvement. We encourage future research to extend language coverage, develop more diverse evaluation metrics, and carefully monitor potential misuses of increasingly capable code generation systems.