CodeAssistBench (CAB): Dataset & Benchmarking for Multi-turn Chat-Based Code Assistance

Myeongsoo Kim
AWS AI Labs
mysoo@amazon.com

Shweta Garg
AWS AI Labs
shwegarg@amazon.com

Baishakhi Ray
AWS AI Labs
rabaisha@amazon.com

Varun Kumar
AWS AI Labs
kuvrun@amazon.com

Anoop Deoras
AWS AI Labs
adeoras@amazon.com

Abstract

Programming assistants powered by large language models have improved dramatically, yet existing benchmarks still evaluate them in narrow code-generation settings. Recent efforts such as InfiBench and StackEval rely on Stack Overflow questions and remain limited to single-turn interactions, manually curated data, and isolated snippets rather than full project environments. We introduce CodeAssistBench (CAB), the first benchmark for evaluating multi-turn, project-grounded programming assistance at scale. CAB automatically constructs datasets from GitHub issues tagged as questions, using an LLM-driven pipeline that filters noise, extracts runnable contexts, builds executable containers, and verifies environment correctness. This enables continuous, automated expansion across diverse repositories without manual intervention. Using CAB, we create a testbed of 3,286 real-world issues across 214 repositories, spanning seven languages. Evaluating state-of-the-art models reveals a substantial gap: while models achieve 70–83% accuracy on Stack Overflow–style questions, they solve only 7.22–16.49% of CAB issues from post-training-cutoff repositories. These results highlight a fundamental challenge: current LLMs struggle to provide assistance in realistic, project-specific contexts despite strong performance on traditional Q&A benchmarks. CAB provides a scalable, reproducible framework for advancing research in multi-turn, codebase-grounded programming agents. The benchmark and pipeline are fully automated and publicly available at https://github.com/amazon-science/CodeAssistBench/.

1 Introduction

Large language models (LLMs) are increasingly integrated into modern programming workflows, supporting tasks ranging from code generation to debugging and repository navigation. While benchmarks have progressed from isolated synthesis tasks (HumanEval Chen et al. [2021], MBPP Austin et al. [2021]) to repository-level maintenance settings (SWE-Bench Jimenez et al. [2023], BigCodeBench Zhuo et al. [2024]), they still capture only a narrow slice of how developers seek assistance in practice. Recent multi-turn benchmarks such as ConvCodeWorld Han et al. [2025], MINT Wang et al. [2023], and TICODER Fakhoury et al. [2024] move toward conversational evaluation, yet they primarily emphasize code synthesis and assume stable, well-defined contexts.

However, real-world programming assistance extends far beyond writing code. A 2024 Stack Overflow survey of 34,168 developers reports that while 75.7% seek AI for code generation, even higher percentages desire support for searching for answers (77.9%), debugging and troubleshooting (77.3%), and understanding unfamiliar codebases (73.6%) Stack Overflow [2024]. These tasks

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Track on Datasets and Benchmarks.

 https://doi.org/10.52202/085713-4280

require iterative clarification, environment-aware reasoning, and integration with project-specific details—capabilities that current benchmarks largely overlook.

Recent efforts such as InfiBench Li et al. [2024] and StackEval Shah et al. [2024] incorporate Stack Overflow data to better reflect developer Q&A scenarios, but they remain limited to single-turn interactions over isolated snippets and require substantial manual curation.

Real assistance is inherently interactive. Even seemingly simple questions often require the assistant to infer project topology, reference environment configuration, or reconcile ambiguous user descriptions. Figure 1 illustrates a real GitHub issue in which a user asks about container port mapping. Solving it requires the assistant to (1) interpret the repository's network structure, (2) explain that the proxy port is internal and hard-coded, and (3) assure the user that no additional mapping is needed. Each answer influences the next question, defining a trajectory of reasoning and communication. Evaluating such scenarios cannot rely solely on execution oracles or single-shot correctness; it requires tracking how well the assistant guides the user across turns with clarity, efficiency, and accuracy.

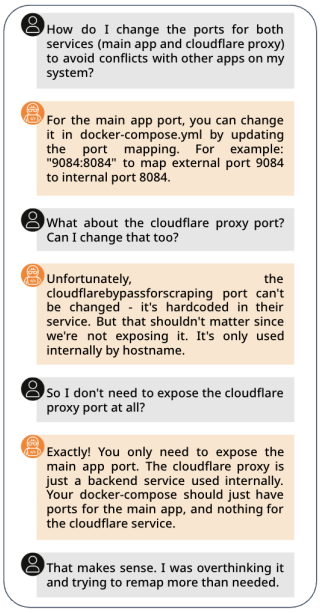

Figure 1: A port-mapping clarification from a real GitHub issue.

To address these limitations, we introduce **CodeAssistBench (CAB)**, the first benchmark designed to evaluate multi-turn, project-grounded programming assistance at scale. CAB automatically constructs datasets from GitHub issues tagged as questions using an LLM-driven pipeline that filters noisy issues, extracts runnable contexts, builds containerized environments, and verifies their correctness. This automation enables continuous, reproducible benchmark expansion without manual curation and supports dataset variants drawn from repositories created after major model training cutoffs, ensuring persistent difficulty over time.

CAB consists of 3,286 real-world programming questions across 214 repositories in seven languages, paired with an execution-backed, chat-driven evaluation pipeline. A simulated user provides contextual feedback, a maintainer agent interacts with the full codebase in a containerized environment, and an automated judge assesses conversation quality using extracted satisfaction conditions. This framework captures the full complexity of real-world developer–assistant interactions: ambiguous problem descriptions, evolving hypotheses, and multi-turn negotiation of understanding.

Our evaluation of leading LLMs reveals a striking capability gap. While recent models achieve 70–83% accuracy on Stack Overflow-style questions, they solve only 16.49% of CAB issues from post-training-cutoff repositories. These results highlight a fundamental challenge: despite impressive performance on traditional Q&A benchmarks, LLMs struggle to provide reliable assistance in realistic, multi-turn, project-specific contexts.

CAB provides a scalable and reproducible foundation for advancing research in multi-turn programming assistance. By coupling fully automated dataset generation with environment-grounded conversational evaluation, it offers a new lens through which to measure and improve the real-world capabilities of LLM-based developer tools.

2 Related Work

Although programming assistance spans diverse developer needs, existing benchmarks only capture narrow aspects of real-world workflows.

Code Generation Benchmarks. Traditional evaluations of programming assistants focus on generating functionally correct code from well-specified prompts. HumanEval Chen et al. [2021], MBPP Austin et al. [2021], and CodeContests Li et al. [2022] assess single-shot code synthesis, while more advanced benchmarks such as SWE-Bench Jimenez et al. [2023] and SWE-PolyBench Rashid et al. [2025] use real GitHub issues to evaluate repository-level maintenance tasks. These benchmarks

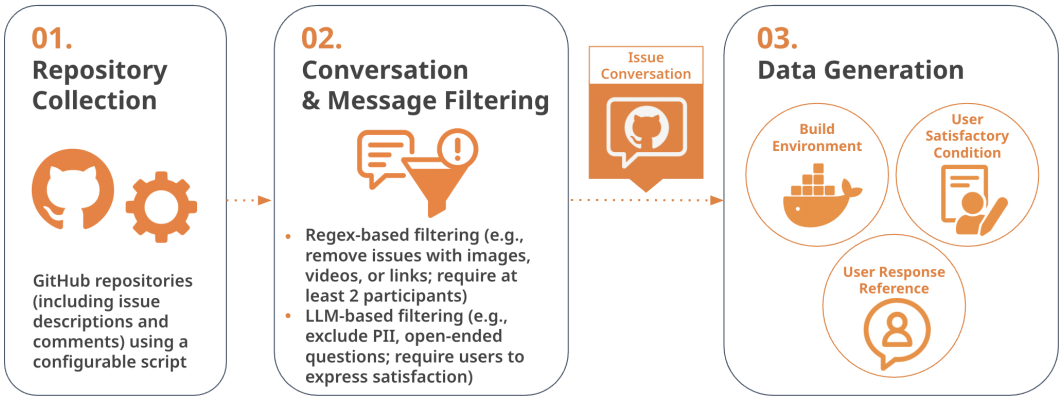

Figure 2: CAB's automated data generation pipeline. It collects relevant GitHub repositories, filters issue conversations, and produces structured assistance scenarios with build environments, satisfaction conditions, and reference user responses.

provide robust execution-based correctness metrics but remain predominantly single-turn and do not capture the iterative, conversational nature of real developer interactions.

Multi-turn Programming and Conversational Benchmarks. Several benchmarks incorporate iterative interactions. ConvCodeWorld Han et al. [2025], MINT Wang et al. [2023], and TICODER Fakhoury et al. [2024] simulate developer–assistant dialogues but typically assume stable, fully specified environments and remain centered on code synthesis tasks. More general conversational evaluations—such as MT-Bench Zheng et al. [2023], AlpacaEval Dubois et al. [2023], and BotChat Duan et al. [2024]—measure dialogue quality but are not targeted at technical problem-solving in programming contexts. In contrast, we evaluate multi-turn, task-grounded interactions where assistance depends on project state, ambiguous context, and evolving user needs.

Programming Q&A Benchmarks. StackEval Shah et al. [2024] and InfiBench Li et al. [2024] use Stack Overflow data to better reflect practical developer questions and emphasize explanatory reasoning. However, they remain limited to single-turn settings, require substantial manual curation, and lack integration with executable project environments. These constraints prevent them from capturing the iterative reasoning, context reconstruction, and environment-aware troubleshooting that dominate real-world programming assistance.

Overall, prior work evaluates code generation, multi-turn dialogue, or Q&A reasoning in isolation. **CAB is the first benchmark to combine multi-turn conversational evaluation with full-project, environment-grounded contexts generated entirely automatically from real GitHub workflows.**

3 CAB: CodeAssistBench

CodeAssistBench (CAB) is a fully automated benchmark framework for evaluating multi-turn, project-grounded programming assistance. CAB converts real GitHub issues into structured, executable assistance scenarios and evaluates LLMs through simulated multi-agent interactions. The framework has two components: (1) an automated dataset construction pipeline and (2) an environment-grounded conversational evaluation system.

3.1 Dataset Generation Pipeline

CAB transforms raw GitHub issues into structured assistance tasks through three stages: (1) repository collection, (2) issue filtering, and (3) data preparation. An overview is shown in Figure 2.

3.1.1 Repository Collection

We begin by collecting a diverse set of GitHub repositories that represent real-world programming challenges across multiple languages. Let $\mathcal{R}_{GH}$ denote the set of all public GitHub repositories. We extract a filtered subset:

$$R = \{r \in \mathcal{R}_{GH} \mid s(r) > S_{\min},\ t(r) > t_0,\ \ell(r) \in \mathcal{L}\} \tag{1}$$

https://doi.org/10.52202/085713-4280

where $s(r)$ is the star count of repository r with threshold $S_{\min}$ (e.g., 10), $t(r)$ is its creation date with cutoff t_0 (e.g., `2024-11-01`), and $\ell(r)$ is r's SPDX license, restricted to permissive licenses compatible with research use.

To prioritize repositories with active developer support communities, we define a community score $\mathrm{CS}(r) = Q(r) + H(r)$, where $Q(r)$ and $H(r)$ represents the number of closed issues labeled with "question" and "help wanted" respectively. We then select the top-N repositories ranked by $\mathrm{CS}(r)$ (breaking ties by star count) to form our repository set R_N, where N is a user-defined parameter.

3.1.2 Issue Filtering

For each repository $r \in R_N$, let I_r be the set of issues associated with it. We fetch only successfully closed issues using the GitHub Issues API. Each issue is represented as:

$$i = (\mathrm{title}(i),\ \mathrm{body}(i),\ \mathrm{messages}(i)), \quad \mathrm{messages}(i) = (m_{i,1}, m_{i,2}, \dots) \tag{2}$$

We apply two filtering rules - implemented using regular expressions - to retain high-quality, interactive conversations: (1) requiring at least two distinct participants to ensure genuine question-answering dynamics; and (2) removing issues containing media content (e.g., URLs, images, videos) to focus on text-based programming assistance.

To ensure issue relevance and message quality, we apply two LLM-based filtering steps using structured prompts. At the issue level, we assess resolution status, specificity, clarity, and safety via a seven question prompt (see Listing A.1), asking the model to answer each of them with a binary Yes/No. Issues are retained only if they satisfy criteria such as being clearly resolved, technically specific, free of sensitive content (e.g., personal information), and reproducible. At the message level, we construct the full conversation and prompt the model to identify comments that provide no support-related value (e.g., "+1", "Thanks", "Bump"). Comments are preserved unless explicitly flagged for removal using strict exclusion rules (see Listing A.2).

After filtering, we restructure each issue as a sequence of turns, where a turn represents a complete user-maintainer interaction:

$$\mathrm{turn}_{i,k} = (m_{i,k}^{\mathrm{author}}, m_{i,k}^{\mathrm{maintainer}}) \tag{3}$$

To construct turns, we group consecutive messages from the same role (author or maintainer) into a single logical message, then pair each author segment with the subsequent maintainer segment. Unpaired trailing messages (e.g., a final "thanks" from the author) are discarded.

The resulting filtered issue set I'_r contains issues with this turn-based structure:

$$i = (\mathrm{title}(i),\ \mathrm{body}(i),\ \{\mathrm{turn}_{i,1}, \mathrm{turn}_{i,2}, \dots\}) \tag{4}$$

3.1.3 Data Preparation

For each filtered issue, we prepare three essential components: build environment generation, user satisfaction condition extraction, and user response reference generation.

Build Environment Generation. For issues requiring environment-specific testing, we automatically generate Docker configurations by analyzing repository artifacts using Sonnet 3.7 Anthropic [2025] with a structured prompt (Appendix A.3) that identifies the commit sha_i closest to the issue creation timestamp, clones the repository at that commit, and extracts key artifacts (README content, Dockerfiles, GitHub workflows, file structure). It then generates and tests candidate build scripts until finding a successful configuration e_i.

Satisfaction Condition Extraction. We use Sonnet 3.7 with a structured prompt (provided in Appendix A.4) to identify explicit criteria that indicate when an issue is successfully resolved. The model analyzes the full conversation, focusing on the original question and subsequent clarifications, to extract concrete conditions that would satisfy the user's needs. These form the set $s_i = \{s_{i,1}, \dots, s_{i,K}\}$ of satisfaction conditions that serve as objective evaluation criteria.

User Response Reference Generation. To enable realistic simulation of user follow-up behavior, we construct a BM25 index over historical maintainer-user message pairs (excluding data from the current issue) and retrieve the top-N most similar maintainer responses with their corresponding user replies. These form the reference set $u_i = \{u_{i,1}, \dots, u_{i,M}\}$ that guides the simulated user's feedback.

Result Aggregation. The resulting dataset entry for each issue is a tuple $d_i = (r_i,\ i,\ sha_i,\ e_i,\ s_i,\ u_i)$. For each issue $i \in \bigcup_r I'_r$, where I'_r denotes the filtered issues for repository r, we create such entries to form our complete dataset.

3.2 Evaluation Framework and Implementation

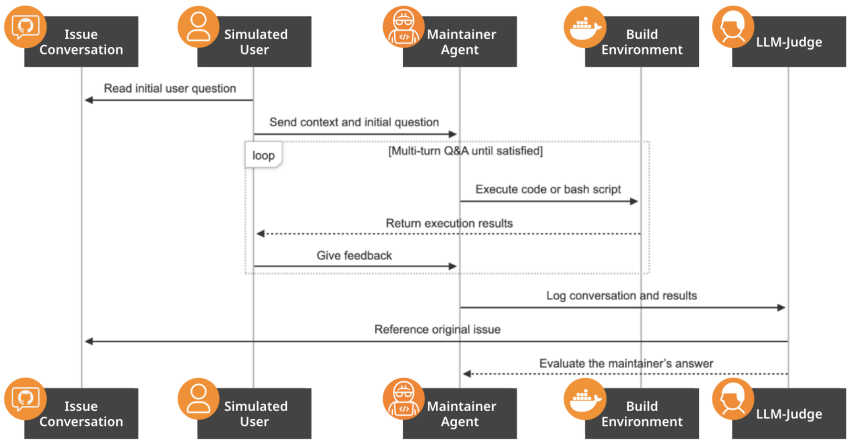

Figure 3: **CAB evaluation pipeline.** A simulated user chats with the Maintainer Agent, which can run code in an optional build sandbox; the interaction continues until the user is satisfied or reaches the maximum turn limit. Once the dialogue ends, an LLM judge grades the exchange against satisfaction conditions extracted from the original GitHub issue. This pipeline enables realistic assessment of programming assistance in context-rich, project-specific scenarios.

CAB's evaluation framework simulates realistic programming assistance interactions through a multi-agent system with three distinct roles: a User, a Maintainer, and a Judge (see Figure 3).

User Agent. The user agent initiates the conversation with a programming question from a GitHub issue and provides follow-up responses. It presents the initial question with context, evaluates model responses against satisfaction conditions, provides realistic follow-ups or clarifications, and signals when a solution resolves the issue. The user agent observes execution results but does not directly interact with the environment.

Maintainer Agent. The maintainer agent represents the LLM being evaluated. Given the user's question and contextual information, the model must analyze the problem within the provided codebase context, execute commands in the containerized environment when necessary, generate helpful responses, and adapt its approach based on user feedback.

Judge Agent. After the conversation concludes, an automated judge evaluates the interaction quality based on: (1) technical correctness - whether the proposed solution correctly addresses the underlying issue; (2) satisfaction completeness - whether all extracted satisfaction conditions are met; and (3) interaction quality - whether the conversation was appropriately concise and helpful. For issues with Docker environments, execution success is a hard requirement - a technically sound explanation is insufficient if the implementation fails in practice. The evaluation is triggered when either the user expresses satisfaction or the conversation reaches a maximum of 10 turns (configurable). This approach provides a comprehensive assessment of an LLM's ability to provide effective programming assistance in realistic scenarios.

CAB is implemented as a fully automated pipeline that continuously generates new benchmark datasets as GitHub repositories evolve. Our implementation integrates with GitHub's REST API to collect repositories and issues with configurable parameters, automatically generates and validates Docker environments to ensure reproducibility, and orchestrates the multi-agent conversation flow while recording detailed interaction logs. This enables CAB to generate continuously evolving benchmarks that remain challenging as models improve, particularly through dataset variants including repositories created after model training cutoff dates.

https://doi.org/10.52202/085713-4280

Table 1: Summary of issue filtering, Docker requirement detection, build outcomes, and final issue retention across programming languages and repository cohorts in our dataset generation pipeline. The 56-issue difference between filtered issues (3,342) and final retained issues (3,286) corresponds to Docker-required issues where environment builds failed (294 Docker-required − 238 successful builds = 56 excluded).

Metric	Total	Python	Java	C++	C#	JS	TS	C
Filtered Issues	3,342	660	543	518	545	445	443	188
All-Time	3,033	500	535	487	520	406	417	168
Recent	309	160	8	31	25	39	26	20
Docker-Required	294	90	58	53	30	34	12	17
All-Time	252	64	58	48	28	30	9	15
Recent	42	26	0	5	2	4	3	2
Successful Docker Builds	238	77	52	28	21	33	10	17
All-Time	197	52	52	23	19	29	7	15
Recent	41	25	0	5	2	4	3	2
Final Retained Issues	3,286	647	537	493	536	444	441	188
All-Time	2,978	488	529	462	511	405	415	168
Recent	308	159	8	31	25	39	26	20

4 Experimental Results and Analysis

We evaluate CAB along three dimensions: (1) dataset generation outcomes, (2) model performance in multi-turn debugging, and (3) human studies validating judge reliability and satisfaction-condition quality. Hardware details are provided in Appendix E.

4.1 Statistics of Dataset Generation Pipeline

We applied our automated pipeline (Section 3.1) to two repository cohorts: an **All-Time** set of 700 top-starred repositories (100 per language) without creation-date limits, and a **Recent** set of 3,500 repositories (500 per language) created after November 2024. After applying license and community filters, we retain 770 repositories for downstream processing. These produced 25,656 raw GitHub issues tagged with `question` or `help-wanted`. We apply a layered filtering process, regex heuristics followed by an LLM classifier, to remove low-signal or ambiguous issues. Issues requiring Docker environments are excluded if automated build validation fails.

Across both cohorts, we retained **3,286 multi-turn issues from 214 repositories**, including **238 fully validated Docker environments**. Many repositories yielded no qualifying multi-turn issues after filtering, resulting in 214 repositories contributing at least one benchmark instance. Notably, **Recent repositories achieved a 97.6% build success rate** compared to 78.2% for All-Time repositories (details in Appendix F). In total, our pipeline executed 44,628 Sonnet 3.7 calls end-to-end without any manual intervention. Issue-length distributions (Figure 5) show that over half of all issues involve multi-turn threads, particularly in Python, C++, and TypeScript—highlighting the need for multi-step reasoning benchmarks such as CAB.

4.2 Model Evaluation Results

We benchmark six state-of-the-art LLMs as maintainer agents: ChatGPT 4.1 Mini, DeepSeek R1, Llama 3.3 70B, Haiku 3.5, Sonnet 3.7, and Sonnet 3.7 (Think). Models interact with simulated users until all satisfaction conditions are met or the turn limit is reached.

4.2.1 Evaluation Metrics

We measure:

- **Correctness**: *Correct*, *Partially Correct*, or *Incorrect*, determined by a judge LLM evaluating condition satisfaction, execution outcomes, and reference alignment.
- **Verbosity**: *Appropriate*, *Verbose*, or *Terse*, capturing explanation style and conciseness.

Judge prompts and scoring criteria are in Appendix A.5.

Table 2: Correctness and average CAB turns. Rec. = Recent repositories; All = All-Time.

Model	Correctness (%)						Avg. CAB Turn					
	Correct		Partially Corr.		Incorrect		Correct		Partially Corr.		Incorrect	
	Rec.	All	Rec.	All	Rec.	All	Rec.	All	Rec.	All	Rec.	All
ChatGPT 4.1 Mini	**16.49**	**29.14**	30.41	36.86	**53.09**	34.00	2.94	2.35	3.66	3.33	5.70	4.28
DeepSeek R1	11.34	27.14	33.51	40.29	55.15	**32.57**	2.82	2.24	3.18	2.72	4.50	4.28
Llama 3.3 70B	9.33	13.58	25.91	40.75	64.77	45.66	3.22	2.68	4.06	3.49	4.67	4.50
Haiku 3.5	7.22	16.86	30.93	43.43	61.86	39.71	3.86	2.73	3.97	3.81	6.76	5.63
Sonnet 3.7	11.34	25.71	30.93	35.43	57.73	38.86	2.36	2.30	3.57	3.15	5.71	4.21
Sonnet 3.7 (Think)	13.40	27.43	27.32	38.86	59.28	33.71	2.50	2.20	3.25	2.85	4.95	4.26

4.2.2 Sampling Strategy

Running all 3,286 issues in multi-turn mode is computationally prohibitive. We therefore construct a balanced subset: per language, up to five Dockerfile issues plus ten issues per turn length bucket (1, 2, 3, 4, 5+). Underflow buckets reallocate to earlier ones. This yields **350 datasets** for All-Time repositories and **194 datasets** for Recent repositories.

4.2.3 Correctness Results

Table 2 presents the performance metrics and conversation statistics for all evaluated models. We observe a consistent performance gap between recent and historical repositories. On recent repositories, correctness rates range from 7.22% (Haiku 3.5) to 16.49% (ChatGPT 4.1 Mini), while on all-time repositories, these rates increase substantially to 13.58-29.14%. ChatGPT 4.1 Mini demonstrates superior performance across both datasets, achieving the highest correctness (16.49% recent; 29.14% all-time) and lowest error rates on recent repositories (53.09%), while DeepSeek R1 shows the lowest error rate on all-time repositories (32.57%).

To better understand the reason behind this recency gap, we investigate two possibilities: (1) model knowledge limitations, and (2) intrinsic properties of newer repositories. To isolate these effects, we construct a lightweight synthetic ablation using 50 automatically generated repositories (details in Appendix J). Surprisingly, Claude 3.7 Sonnet achieved 74% correctness on these synthetic repositories compared to only 11.34% on recent real repositories. The synthetic repositories contained richer documentation and version constraints within model training windows, suggesting that the recency gap is primarily driven by post–knowledge-cutoff framework changes rather than properties of AI-generated code. Full methodology and analysis appear in Appendix J.

4.2.4 Qualitative Example

Figure 4 illustrates a GitHub issue where the user asks how to remap ports for both the main application and the Cloudflare proxy to avoid conflicts with other apps. It presents two contrasting model *stories*: Haiku 3.5 provides an incomplete response, either omitting clear proxy-port guidance or introducing incorrect configurations, while ChatGPT 4.1 Mini offers a successful solution - clearly explaining how to change the main app's port, noting that the proxy port doesn't need to be exposed, and providing a conflict-free `docker-compose.yml`. A simulated user agent drives each LLM through a multi-turn dialogue, asking follow-up questions until all satisfaction conditions are met or a predefined turn limit is reached. Where applicable, the agent also validates the recommended configuration using recorded build outcomes. Once the conversation concludes, an automated LLM judge assigns a final label - *Correct*, *Partially Correct*, or *Incorrect* - based on comparisons with the original GitHub thread, execution results, and the predefined satisfaction conditions.

4.2.5 Conversation-Length Analysis

Table 2 compares the average turn counts between original GitHub threads and CAB-generated dialogues. When models produce correct answers, CAB conversations are similar in length to real GitHub threads - typically spanning 2–3 turns. In contrast, incorrect cases tend to result in longer CAB dialogues, often by 1–2 additional turns. This suggests that when the model-as-maintainer provides vague or incomplete responses, the simulated user - like a real developer - asks follow-up questions for clarification.

 https://doi.org/10.52202/085713-4280

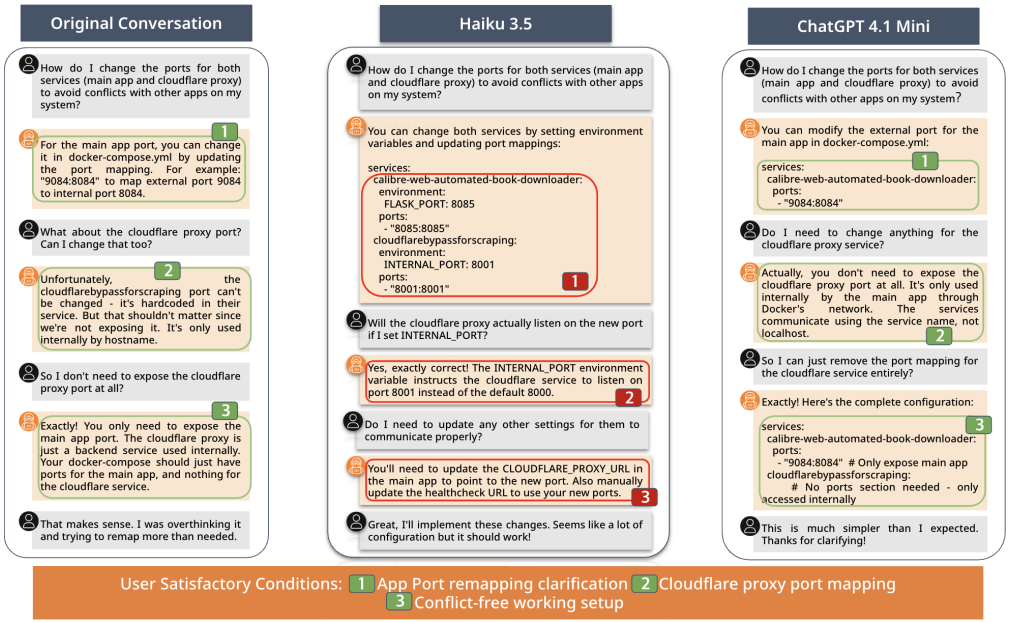

Figure 4: Side-by-side comparison of model responses to a Docker port-remapping issue: Haiku 3.5's incomplete solution (middle) fails to address key requirements, while ChatGPT 4.1 Mini's successful response (right) satisfies all three user conditions (highlighted in colored boxes).

4.2.6 Verbosity Analysis

Figure 7 (Appendix) summarizes verbosity classifications. Models tend toward over-explanation: verbose responses account for 40–60% of outputs. Sonnet 3.7 (Think) shows the most balanced behavior, while Haiku 3.5 and Llama 3.3 70B are most verbose. Verbosity increases on recent repositories, likely due to higher uncertainty.

4.2.7 Language-Specific Analysis

Figure 6 reports the percentage of *Correct* responses - excluding partially correct answers - for each model across seven programming languages, split by recent and all-time GitHub repositories. The analysis reveals stark contrasts in language difficulty and model specialization.

Statically typed languages such as C#, C++, and Java remain particularly challenging, especially on recent issues. For instance, in the recent C# dataset, most models achieve less than 13% correctness. Similarly, correctness on recent C++ issues hovers below 15% for all models except ChatGPT 4.1 Mini. These results suggest models often struggle with precision and strict type constraints in newer repositories.

By contrast, dynamically typed languages like JavaScript and Python show relatively stronger performance on the all-time dataset. ChatGPT 4.1 Mini and Sonnet 3.7 Think both reach 44% correctness on JavaScript, while DeepSeek R1 and Sonnet 3.7 Think achieve over 30% on Python and TypeScript. Nevertheless, performance on recent repositories remains low across the board. Even for JavaScript, the best model - Sonnet 3.7 Think - achieves only 15.4% correctness. These results highlight the increased complexity or novelty of recent code issues.

Overall, Sonnet 3.7 Think (M6) consistently ranks among the top performers in JavaScript, TypeScript, and Python. ChatGPT 4.1 Mini (M1) also shows strength in all-time datasets for JavaScript and TypeScript. No model, however, maintains high correctness across both typed and dynamic languages, particularly in recent repositories.

4.3 Human Evaluation Studies

We conducted two human evaluation studies to validate CAB's automated components: judge reliability and satisfaction condition quality.

4.3.1 Judge Validation

To assess the reliability of our automated LLM judge, two software engineers (3+ years experience) independently evaluated 310 model responses sampled across languages, difficulty levels, and issue types. The annotators achieved substantial inter-rater agreement (78.28%, Cohen's $\kappa = 0.68$; see Table 4 in Appendix B), establishing a strong human baseline.

Comparing the automated judge to human annotators, the LLM judge achieved 65.92% average agreement—84.2% of the inter-human baseline. Agreement was highest on objective dimensions such as verbosity (93.2% of human-level agreement) and lower on subjective dimensions such as technical correctness (86.7% of human-level agreement). Full methodology, annotation protocols, and judge–human comparison details are provided in Appendix B.

4.3.2 Error Category Analysis

To better characterize how models fail across different problem types, we categorized the 310 cases from our human annotation study (Section B) into seven coarse error types using LLM-assisted classification with Sonnet 3.7. Logic errors (28.8%) and environment configuration issues (28.5%) dominate the dataset, while performance-related issues exhibit the highest annotator–judge agreement (75.0%). The full error taxonomy, per-model breakdowns, and implications for future benchmark design are provided in Appendix K.

4.3.3 Satisfaction Condition Validation

We evaluated automatically extracted satisfaction conditions through annotations by 21 professional contractors (mean 5.1 years programming experience). Across 663 conditions from 70 randomly sampled issues, 86.3% were judged accurate but only 65.7% complete, indicating high precision with conservative recall. Our pipeline favors reliable extractions over exhaustive coverage. Detailed annotation guidelines and per-language statistics are in Appendices I and H.

5 Limitations

While CodeAssistBench (CAB) offers a realistic and scalable framework for evaluating multi-turn programming assistance, several limitations remain, including conservative condition extraction, limited evaluation coverage, templated user behavior, and language scope constraints. Additionally, our LLM-based judge achieves 65.92% agreement with human annotators (84.2% of the 78.28% inter-human baseline), suggesting multi-turn evaluation introduces alignment challenges compared to single-turn settings; see Appendix C for full discussion.

6 Conclusion and Future Work

Our study introduces CodeAssistBench (CAB), a fully automated benchmark for evaluating multi-turn programming assistance grounded in real-world developer interactions. Unlike prior benchmarks that emphasize single-turn or synthetic tasks, CAB simulates realistic conversations over full codebases using containerized environments and issue-specific satisfaction criteria. Through extensive experiments, we find that state-of-the-art language models struggle with complex, multi-turn dialogues - particularly on recent repositories. CAB's automated pipeline enables scalable dataset expansion while supporting rigorous, reproducible evaluation across languages, project types, and time periods, offering a more faithful measure of model capabilities than existing benchmarks.

Several promising directions could extend CAB's impact: improving satisfaction-condition extraction to balance precision and recall, expanding language coverage beyond the current seven languages, incorporating more sophisticated user-simulation strategies, and designing specialized metrics for assistance categories (Appendix L). We hope CAB enables deeper insights into current system limitations and guides the development of future AI assistants that more effectively support real-world software engineering workflows. We further discuss potential societal benefits and risks in Appendix D.

 https://doi.org/10.52202/085713-4280

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

Hojae Han, Seung-won Hwang, Rajhans Samdani, and Yuxiong He. Convcodeworld: Benchmarking conversational code generation in reproducible feedback environments. *arXiv preprint arXiv:2502.19852*, 2025.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. Mint: Evaluating llms in multi-turn interaction with tools and language feedback. *arXiv preprint arXiv:2309.10691*, 2023.

Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. Llm-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Trans. Softw. Eng.*, 50(9):2254–2268, September 2024. ISSN 0098-5589. doi: 10.1109/TSE.2024.3428972. URL https://doi.org/10.1109/TSE.2024.3428972.

Stack Overflow. Stack overflow developer survey 2024 – ai tools next year, 2024. URL https://survey.stackoverflow.co/2024/ai#2-ai-tools-next-year. Accessed: 2025-05-13.

Linyi Li, Shijie Geng, Zhenwen Li, Yibo He, Hao Yu, Ziyue Hua, Guanghan Ning, Siwei Wang, Tao Xie, and Hongxia Yang. Infibench: Evaluating the question-answering capabilities of code large language models. *Advances in Neural Information Processing Systems*, 37:128668–128698, 2024.

Nidhish Shah, Zulkuf Genc, and Dogu Araci. Stackeval: Benchmarking llms in coding assistance. *Advances in Neural Information Processing Systems*, 37:36976–36994, 2024.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, Anoop Deoras, Giovanni Zappella, and Laurent Callot. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents, 2025. URL https://arxiv.org/abs/2504.08703.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-bench and chatbot arena. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. URL https://openreview.net/forum?id=uccHPGDlao.

Yann Dubois, Xuechen Li, Rohan Taori, Tianyi Zhang, Ishaan Gulrajani, Jimmy Ba, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Alpacafarm: A simulation framework for methods that learn from human feedback, 2023.

Haodong Duan, Jueqi Wei, Chonghua Wang, Hongwei Liu, Yixiao Fang, Songyang Zhang, Dahua Lin, and Kai Chen. BotChat: Evaluating LLMs' capabilities of having multi-turn dialogues. In Kevin Duh, Helena Gomez, and Steven Bethard, editors, *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 3184–3200, Mexico City, Mexico, June 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-naacl.201. URL https://aclanthology.org/2024.findings-naacl.201/.

Anthropic. Claude 3.7 sonnet. https://www.anthropic.com, 2025. Accessed May 2025.

J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.

Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. Automated test generation for rest apis: No time to rest yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 289–301, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393799. doi: 10.1145/3533767.3534401. URL https://doi.org/10.1145/3533767.3534401.

Myeongsoo Kim, Davide Corradini, Saurabh Sinha, Alessandro Orso, Michele Pasqua, Rachel Tzoref-Brill, and Mariano Ceccato. Enhancing rest api testing with nlp techniques. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 1232–1243, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702211. doi: 10.1145/3597926.3598131. URL https://doi.org/10.1145/3597926.3598131.

Myeongsoo Kim, Tyler Stennett, Dhruv Shah, Saurabh Sinha, and Alessandro Orso. Leveraging large language models to improve rest api testing. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER'24, page 37–41, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705007. doi: 10.1145/3639476.3639769. URL https://doi.org/10.1145/3639476.3639769.

Myeongsoo Kim, Tyler Stennett, Saurabh Sinha, and Alessandro Orso. A multi-agent approach for rest api testing with semantic graphs and llm-driven inputs. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, page 1409–1421. IEEE Press, 2025a. ISBN 9798331505691. doi: 10.1109/ICSE55347.2025.00179. URL https://doi.org/10.1109/ICSE55347.2025.00179.

Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. Llamaresttest: Effective rest api testing with small language models. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025b. doi: 10.1145/3715737. URL https://doi.org/10.1145/3715737.

A LLM Filtering Prompts

A.1 Issue-Level Relevance Filter

To identify issues suitable for benchmarking, we use a 7-question prompt to assess resolution status, specificity, clarity, and safety. Below is the exact prompt provided to the LLM:

```
Please evaluate the following GitHub issue and its comments:

    Title: {title}

    Author: {author}

    Body:
    {body}

    Comments:
    {comments}
```

```
    Based on this conversation, please answer the following
    questions with Yes or No:
    1. Is the problem resolved by someone other than the
    author (not self-answered)?
    2. Does the conversation contain confirmation from the
    author that the problem has been resolved?
    3. Is the problem a specific technical issue (not a
    feature request, opinion, or open-ended question)?
    4. Is there a clear, definitive solution provided
    within the conversation?
    5. Can the solution be directly applied without
    requiring additional context or resources?
    6. Does the conversation contain any personally
    identifiable information (PII) such as Email addresses,
    Phone numbers, Physical addresses, Full names (beyond
    just GitHub usernames), Passwords or credentials,
    Personal identification numbers, IP addresses, or Any
    other sensitive personal information?
    7. Can this problem be reproduced and solved using the
    provided solution today (April 2025)?

Please provide your answers in the format:
    1. [Yes/No]
    2. [Yes/No]
    3. [Yes/No]
    4. [Yes/No]
    5. [Yes/No]
    6. [Yes/No]
    7. [Yes/No]
```

The model responds with binary answers to each question, which we then parse to determine whether the issue is usable.

A.2 Message-Level Comment Filter

For message filtering, we construct the full conversation context and ask the LLM to identify non-contributory messages (e.g., "+1", "Thanks", "Bump"). A comment is retained unless explicitly marked for removal. The prompt follows:

```
Analyze each comment in this GitHub issue conversation:

{conversation}

Your task is to identify ONLY comments that have ABSOLUTELY
    NO support-related value.
A comment should ONLY be removed if it falls into ALL of
    these criteria:
- Contains NO technical information
- Provides NO context about the issue
- Asks NO relevant questions (technical or process-related)
- Provides NO status updates or next steps
- Offers NO feedback on proposed solutions
- Contains NO clarifications about the user's situation or
    environment
- Has NO administrative or process value (like assigning
    work, requesting more info)

Examples of comments to remove:
1. Pure social messages at conversation end: "Thanks!",
    "Cool", "thumbs-up imoji"
2. Empty status updates: "+1", "Same issue", "Any
    updates?", "Bump" with no additional context
3. Completely off-topic discussions unrelated to the issue
```

```
IMPORTANT: Preserve comments that show the natural flow of
    support interaction. If a comment contains ANY
    support-related value, even if minimal or alongside
    thanks/acknowledgements, DO NOT remove it.

List ONLY the comment numbers that should be removed
    because they have absolutely no support-related value.
Format:
NUMBERS: <comma-separated list of numbers>
EXPLANATION: <specific reasons why these comments add no
    support-related value>

If no comments should be removed, respond with:
NUMBERS: none
EXPLANATION: All comments contain some support-related
    value or context
```

A.3 Automated Dockerfile Synthesis and Repair

To ensure that every benchmarked issue can be built and tested in a fully self-contained environment, we automatically *(i)* generate an initial set of Dockerfile candidates and *(ii)* iteratively refine any candidate that fails to build.

Phase 1: Candidate Generation. For each GitHub issue we collect a rich context bundle—repository URL, commit SHA, truncated issue body, `README`, a structure summary of the repo, up to two GitHub Actions workflows, and (if available) a reference Dockerfile from the same project. We then prompt the LLM to produce 5 self-contained Dockerfiles, each of which must (1) install dependencies, (2) clone the repo at the specified commit, and (3) build the project without executing the user's code. The system prompt fixes the LLM's persona ("expert Docker engineer"), while the user prompt supplies the per-issue context. By requesting *plain text only* (no Markdown fences) we can pipe the response directly to `docker build`.

```
SYSTEM: You are an expert Docker engineer who creates Dockerfiles to
    build and validate GitHub issues.

Repository URL: {repo_url}
Title: {issue_title}
Commit SHA: {commit_sha}

Issue description (truncated to 3 kB):
{issue_body}

Context:
- README
- Repo structure summary
- GitHub workflow files (optional)
- Reference/Original Dockerfile (optional)

Create a Dockerfile that
1. installs all dependencies in the issue,
2. clones the repository and checks out {commit_sha} or user given
    project version,
3. builds the project (no test or run commands)

IMPORTANT: Return only the raw Dockerfile content - no Markdown, no
    commentary.
```

Listing 1: Prompt for candidate generation

Phase 2: Fault-Directed Repair. If every candidate in Phase 1 fails to build, we capture the build log and feed it—together with the failing Dockerfile and the same context bundle—into a repair

 https://doi.org/10.52202/085713-4280

prompt. The LLM is asked to produce an improved Dockerfile that specifically addresses the observed errors. We repeat this loop for up to three attempts or until a build succeeds.

```
SYSTEM: You are an expert Docker engineer who specializes in fixing
    Dockerfiles that failed to build.

USER:
Repository URL: {repo_url}
Issue #: {issue_number}   Title: {issue_title}
Commit SHA: {commit_sha}

Failing Dockerfile:
{candidate_dockerfile}

Build error (truncated to 3 kB):
{build_error}

Provide a corrected Dockerfile that
1. removes the above error(s),
2. keeps the minimal environment needed to build,
3. follows the same constraints as the generation prompt.

IMPORTANT: Return only the raw Dockerfile content - no Markdown, no
    commentary.
```

Listing 2: Prompt for candidate repair

A.4 Satisfaction-Condition Extraction

To evaluate whether an assistant's reply *actually* meets a developer's needs, we first distill each GitHub issue thread into a small set of **user-satisfaction conditions**—explicit criteria that any acceptable answer must fulfill. The extraction procedure is entirely automated and comprises two stages.

Stage 1: LLM-based Extraction. We prompt a language model to extract these conditions from each conversation using two coordinated prompts: a *system prompt* that defines the task, abstraction level, and response format, and a *user prompt* that injects issue-specific content. The model returns a JSON object describing each condition along with a brief explanation.

```
You are an expert at analyzing GitHub issues and extracting
    user satisfaction conditions-the criteria by which any
    answer will be judged.

A satisfaction condition states WHAT the user needs, not
    HOW to implement it.

# Abstraction guide
TOO SPECIFIC - "Use numpy.where(...)"
GOOD LEVEL   - "Vectorized conditional selection"
TOO GENERIC  - "A working solution"

# Must-have properties
1. TRANSFERABLE  (solution-agnostic)
2. VERIFIABLE    (pass/fail is clear)
3. EVIDENCED     (grounded in user utterances)
4. NEED-FOCUSED  (problem, not implementation)

Return exactly this JSON:
{
  "satisfaction_conditions": [
    {
      "condition": "...",
      "explanation": "..."
```

```
    }
  ]
}
Do not wrap the JSON in markdown fences.
```

Listing 3: System prompt for satisfaction-condition extraction

```
Given this GitHub conversation:

Title   : {title}
Author  : {author}
Question: {body}

Comments (chronological, at most 100):
{formatted_comments_json}

Extract every user satisfaction condition.
Remember: they are *criteria*, not solutions.

Output exactly one JSON object as described in the system prompt-no
    extra text.
```

Listing 4: User prompt for satisfaction-condition extraction

Stage 2: Post-hoc Verification. The raw JSON returned by the LLM is parsed and each condition is passed through a lightweight verifier that checks:

1. the text parses as valid JSON;
2. each entry includes both a `condition` and an `explanation`;
3. the explanation quotes or paraphrases evidence found in the thread.

Conditions that fail verification are discarded. Issues with no surviving valid conditions are excluded from the benchmark.

A.5 LLM judge

To assess the quality of the maintainer's answer in each conversation, we employ a structured LLM prompt that simulates a judge agent. The agent is provided with (i) the original GitHub issue (title, body, and comments), (ii) the set of user satisfaction conditions, (iii) the maintainer's answer to be evaluated, and (iv) optional Docker validation results. The LLM is instructed to evaluate the answer along three axes: technical correctness, alignment with user satisfaction conditions, and verbosity.

Evaluation Prompt. The judge agent uses the prompt shown in Listing 5:

```
You are a judge evaluating the maintainer's answer to a
    user's technical question.

Your task is to determine if the maintainer's answer is:
1. TECHNICALLY CORRECT
2. SATISFIES USER CONDITIONS
3. APPROPRIATE VERBOSITY

IMPORTANT: For Docker-related issues:
- The answer must be technically correct AND
- The Docker build/test process must succeed

If Docker validation fails (Success: False), the answer is
    considered INCORRECT.

Provide your evaluation in this format:
```

 https://doi.org/10.52202/085713-4280

```
TECHNICAL CORRECTNESS: [CORRECT / PARTIALLY CORRECT /
    INCORRECT]

ALIGNMENT SCORE: X/Y CONDITIONS MET (Z%)

CONDITION 1: [TRUE / FALSE] <brief description>
...

VERBOSITY ASSESSMENT: [TERSE / APPROPRIATE / VERBOSE]

VERDICT: [CORRECT / PARTIALLY CORRECT / INCORRECT]

KEY ISSUES:
- Issue 1
- Issue 2

REASONING:
Detailed explanation of correctness and alignment.
```

Listing 5: Prompt for LLM judge

Inputs. The judge agent receives:

- The full conversation (title, question body, comments),
- User satisfaction conditions,
- Maintainer's generated answer,
- Docker validation logs (if applicable).

Verdict Criteria. The final judgment is based on:

- **Correct:** Fully accurate and satisfies *all* user conditions.
- **Partially Correct:** Minor technical flaws or partial condition satisfaction.
- **Incorrect:** Major errors, unmet conditions, or failed Docker validation.

Post-Processing. The LLM's output is parsed to extract:

- Technical correctness,
- Number of conditions satisfied,
- Verbosity assessment,
- Final verdict,
- Key issues and reasoning.

B Expert Validation: Detailed Methodology and Results

B.1 Validation Methodology

Dataset Construction. We initially sampled 200 unique GitHub issues from real-world repositories across seven programming languages for expert validation. To ensure annotation quality, we filtered out 24 issues containing non-English content, as all annotators were English speakers, resulting in a final set of 176 unique issues.

We then generated agent responses using two state-of-the-art models for these 176 issues: ChatGPT 4.1 Mini and Claude Sonnet 3.7. Initially, both models generated responses for most issues (134 issues received responses from both models to enable comparative analysis). However, not all AI-generated responses were suitable for human annotation. We excluded responses where the agent output contained unicode character encoding issues or null fields in critical parts, which would have

prevented annotators from accurately evaluating the responses. Specifically, 40 GPT responses and 38 Sonnet responses were excluded due to these data quality issues.

After this filtering, we retained 310 usable agent responses for human evaluation: 154 from ChatGPT 4.1 Mini and 156 from Claude Sonnet 3.7. Each of these 310 responses was independently evaluated by two expert annotators, yielding 620 total annotation records. The final language-wise distribution of validated agent responses per model is shown in Table 3.

Table 3: Language distribution of validated agent responses

Language	ChatGPT 4.1 Mini	Claude Sonnet 3.7
Python	41	42
JavaScript	30	31
C++	25	25
C#	22	21
TypeScript	17	18
C	13	13
Java	6	6
Total	**154**	**156**

Annotation Protocol. Two senior software engineers (Human1 and Human2) with 3+ years of experience each independently evaluated all 310 agent responses. Critically, annotators used the *exact same evaluation prompts and criteria* as the LLM judges to ensure fair comparison and minimize methodological confounds. Each response was evaluated across three dimensions:

- **Technical Correctness**: Does the response accurately address the user's technical question?
- **Verbosity**: Is the response appropriately concise, verbose, or terse?
- **Overall Verdict**: Is the response correct, partially correct, or incorrect?

Annotators worked independently without communication, and each agent response was evaluated by both annotators, yielding 620 annotation records (310 responses $\times$ 2 annotators).

B.2 Inter-Rater Reliability

The two expert annotators demonstrated substantial overall agreement at 78.28% (728/930 judgments across all dimensions; Cohen's $\kappa = 0.68$), establishing a robust baseline for evaluating LLM judge performance. According to established guidelines Landis and Koch [1977], κ values between 0.61–0.80 indicate substantial agreement, confirming that our annotation protocol achieves a high level of consistency. Agreement varied across evaluation dimensions, as shown in Table 4.

Table 4: Inter-rater reliability between Human1 and Human2

Dimension	Agreement	Percentage	Cohen's κ
Verbosity	263/310	84.84%	0.77
Technical Correctness	233/310	75.16%	0.63
Verdict	232/310	74.84%	0.62
Overall	**728/930**	**78.28%**	**0.68**

This variation reveals that while response style (verbosity) is relatively objective and unambiguous ($\kappa = 0.77$, substantial agreement), assessing technical accuracy requires nuanced domain expertise and can yield different but equally defensible judgments ($\kappa = 0.62$–0.63, moderate to substantial agreement). According to established guidelines Landis and Koch [1977], κ values between 0.61–0.80 indicate substantial agreement, and our results fall within this range across all dimensions. This level of agreement demonstrates that even experienced practitioners may weigh correctness, completeness, and stylistic factors differently when evaluating subjective technical tasks.

Our 78.28% agreement rate and Cohen's κ of 0.68 represent substantial inter-rater reliability, validating both our annotation protocol and confirming the inherent subjectivity in evaluating code quality and assistance effectiveness.

 https://doi.org/10.52202/085713-4280

B.3 LLM-Expert Agreement

We compared automated LLM judge evaluations against both expert annotators to assess the reliability of LLM-as-judge for code assistance evaluation. Table 5 summarizes the agreement rates.

Table 5: Agreement rates between LLM judges and expert annotators

Judge Pair	Overall	Technical	Verbosity	Verdict
Human1 vs Human2	78.28%	75.16%	84.84%	74.84%
LLM vs Human1	70.22%	65.16%	79.03%	66.45%
LLM vs Human2	61.61%	53.87%	74.84%	56.13%

The LLM judges achieved 70.22% agreement with Human1, representing 89.7% of the inter-human baseline (78.28%), and 61.61% with Human2 (78.7% of baseline). The average LLM-human agreement of 65.92% reaches 84.2% of the inter-human agreement level, demonstrating that LLM judges approach human-level evaluation capabilities while maintaining scalability advantages. These results reveal several important findings:

Performance on Objective vs. Subjective Metrics. LLM judges achieve their highest agreement with human experts on verbosity assessment (74–79%), where evaluation criteria are most clearly defined and objective. Agreement decreases for technical correctness (54–65%), where deep domain expertise and contextual understanding become critical. The 79.03% LLM-Human1 agreement on verbosity (93.2% of the 84.84% human baseline) demonstrates that automated evaluation excels at surface-level characteristics but requires improvement for nuanced technical assessment.

Variability in Human Standards. The substantial 8.61 percentage point difference in LLM agreement between Human1 (70.22%) and Human2 (61.61%) indicates that individual annotator strictness, expertise emphasis, or judgment criteria significantly impact evaluation outcomes. This variability—comparable to the 8.06 point gap between Human1-LLM (70.22%) and Human1-Human2 (78.28%)—highlights the importance of multi-annotator validation and suggests that single-annotator studies may yield inconsistent conclusions. Our dual-annotator approach provides more reliable ground truth than typical single-expert evaluation.

C Extended Discussion of Limitations

While CodeAssistBench (CAB) provides a realistic and scalable framework for evaluating multi-turn programming assistance, it has several limitations.

First, our satisfaction condition extraction prioritizes precision over recall. Although 86.3% of extracted conditions were judged accurate by human annotators, only 65.7% were complete, suggesting that models may be unfairly penalized for omitting criteria not fully captured by our automated pipeline.

Second, evaluation is performed on a sampled subset of 544 issues (from a pool of 3,286), constrained by computational cost. This may skew results away from rare or edge-case conversations.

Third, the simulated user uses BM25-matched historical responses to simulate follow-ups. While grounded in real-world interactions, this approach may underrepresent the full diversity of developer behaviors, especially in ambiguous or exploratory contexts.

Fourth, CAB only includes issues with successfully synthesized Docker environments. This excludes legacy or atypically configured projects, introducing a bias toward actively maintained, modern repositories with standard build systems.

Fifth, our evaluation targets only seven programming languages and repositories with permissive open-source licenses. The benchmark does not currently assess proprietary codebases, enterprise workflows, or other language ecosystems (e.g., Rust, Go, Swift).

Sixth, we fixed generation temperatures to promote reproducibility: temperature=0 for conversation responses and temperature=0.7 for Dockerfile generation (to encourage diversity across five sampled candidates). While this setup ensures consistency and comparability, it may not reflect each model's optimal decoding parameters for task performance.

Seventh, we do not report error bars or statistical significance tests. While aggregate model trends are clear, fine-grained comparisons—especially between close-performing models—should be interpreted cautiously.

Lastly, the 12.36 percentage point gap between our LLM judge's agreement with human annotators (65.92%) and the inter-human baseline (78.28%) represents a fundamental limitation of automated multi-turn evaluation. This gap is particularly pronounced for subjective dimensions like technical correctness (54–65% LLM-human agreement) compared to objective dimensions like verbosity (74–79%).

We hypothesize that multi-turn conversations introduce several sources of variance that make alignment harder than single-turn evaluation:

1. **Accumulated context interpretation**: As conversations progress, the accumulated context can be interpreted differently by human and LLM judges. A response that seems adequate given early turns may be judged differently when later clarifications reveal the user's true intent.
2. **Credit attribution ambiguity**: In multi-turn dialogues, it becomes unclear which turn's contribution should be credited for ultimate success or failure. Human annotators may attribute success to the final clarifying response, while LLM judges may weight earlier turns more heavily.
3. **Trajectory-dependent evaluation**: The combinatorial space of possible conversation trajectories makes it difficult to establish consistent evaluation criteria. Two conversations reaching the same endpoint via different paths may receive different judgments depending on the intermediate steps.
4. **Error propagation**: Early misunderstandings or incomplete responses can cascade through subsequent turns, making it challenging to isolate which specific failure led to an incorrect final assessment.

This finding has important implications for the broader research community working on LLM-as-judge methods and multi-turn dialogue evaluation. We suggest several directions for future work to address this limitation:

- **Turn-level annotations**: Rather than evaluating entire conversations holistically, decomposing evaluation into per-turn assessments may reduce variance and improve alignment.
- **Trajectory-based metrics**: Developing metrics that explicitly account for conversation dynamics and progression, rather than treating dialogues as atomic units.
- **Consensus-based judging**: Using multiple LLM judges with ensemble methods to reduce individual model biases, similar to how we used multiple human annotators.
- **Calibrated confidence scores**: Training judges to output calibrated confidence scores that reflect uncertainty in multi-turn contexts, enabling downstream systems to weight judgments appropriately.

We leave the systematic investigation of these approaches to future work, but note that improving multi-turn evaluation reliability remains critical for advancing conversational code assistance research.

We believe addressing these limitations—by improving condition recall, scaling human annotation, diversifying simulation strategies, and exploring a broader language and tooling spectrum—will further enhance CAB's utility as a long-term benchmark for conversational coding agents.

D Broader Impacts

CodeAssistBench (CAB) has the potential to advance the development of safer and more capable AI programming assistants, particularly in realistic, multi-turn developer scenarios. By surfacing failure modes and limitations in current models, CAB encourages the creation of tools that better align with developer needs, which may enhance productivity and learning. However, there are also potential negative impacts. For instance, if developers rely too heavily on inaccurate agent suggestions in real-world settings, this could introduce subtle bugs or security issues. Additionally, the use of scraped GitHub data—even when filtered—may raise concerns around privacy or attribution if not

 https://doi.org/10.52202/085713-4280

carefully managed. We encourage responsible use of CAB, including proper consent for dataset extension and monitoring downstream usage of models trained or benchmarked using this framework.

E Hardware Requirements and Runtime Environment

Most stages of the CodeAssistBench (CAB) pipeline—including dataset generation, model evaluation, and human alignment analysis—are driven by API-based interactions with GitHub and commercial LLM endpoints. As a result, these stages require minimal local compute resources and can be executed efficiently on standard cloud instances or modest local machines.

The only hardware-intensive component is the Dockerfile synthesis and validation phase. This stage involves generating candidate Dockerfiles and building them in parallel across hundreds of repositories to ensure correctness and reproducibility. Due to the storage and memory demands of large-scale container builds, we recommend using machines equipped with at least 1TB of local storage and 32GB of memory.

All experiments were conducted on AWS `g5.16xlarge` instances, each with 1 NVIDIA A10G GPU, 64 vCPUs, and 256GB of RAM. While GPU acceleration was not required (as all model queries were served via commercial APIs), the instance's large memory and CPU capacity were beneficial during Docker-based validation. The total estimated compute usage was approximately 100 CPU-hours, with dataset generation taking around 3 days and experimental runs completing in approximately 4 days.

F Docker Build Details

Most build failures in All-Time repositories stemmed from outdated dependencies or unavailable components, not from limitations of the LLM pipeline itself. To address this, CAB inferred project-specific requirements—such as relevant library versions, operating systems, and toolchains—by analyzing both the issue's content and its creation timestamp, then tailored environment reconstruction accordingly via custom Dockerfile generation. The higher success rate in Recent repositories (97.6% vs. 78.2%) reflects better availability of current dependencies and more standardized build configurations in modern projects.

G Supplementary Material

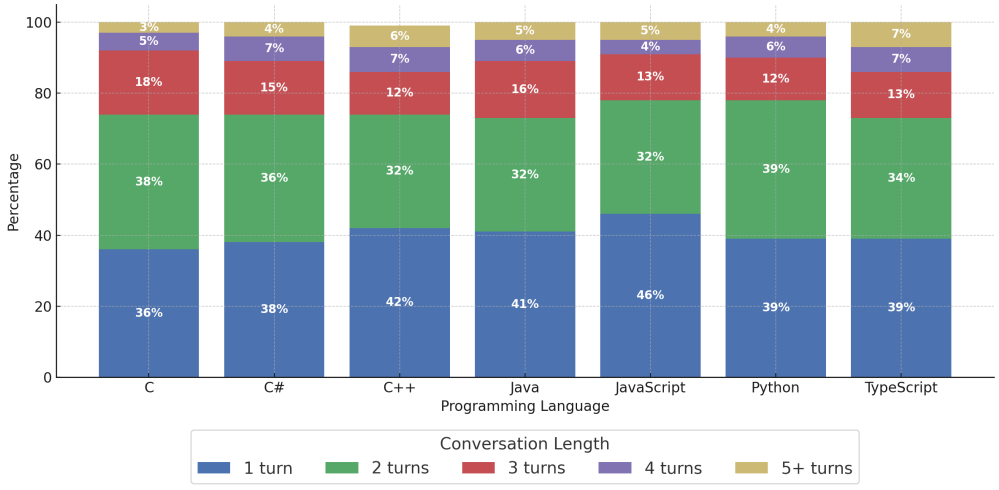

Figure 5: Distribution of GitHub issue conversation lengths by programming language. Each turn corresponds to a maintainer response; for example, a 1-turn conversation consists of a user question and a single maintainer reply, while longer conversations reflect additional back-and-forth exchanges.

Table 6: Docker environment build success rates by programming language

Language	Environment-dependent	Successful Builds	Success Rate
C	17	17	100%
C#	30	21	70%
C++	53	28	53%
Java	58	52	90%
JavaScript	34	33	97%
Python	90	77	86%
TypeScript	12	10	83%
Total	294	238	81%

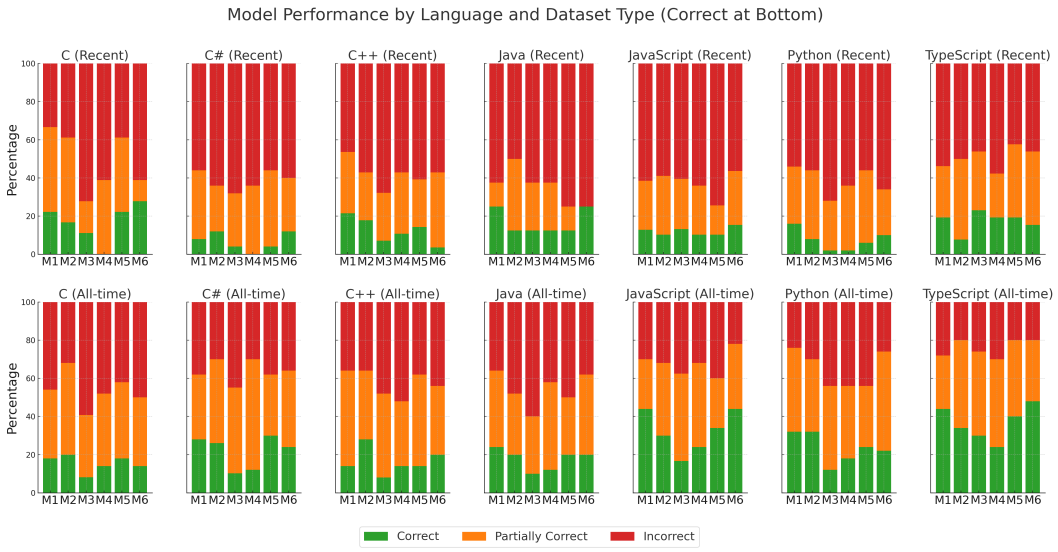

Figure 6: Cumulative comparison of model performance across seven programming languages. Each bar represents a model's prediction outcome distribution on GitHub issues, broken down into **Correct (green)**, **Partially Correct (orange)**, and **Incorrect (red)** responses. The top row corresponds to evaluations on **recent repositories**, while the bottom row shows results on **all-time repositories**. Models are labeled as M1–M6 in the following order: **M1**: ChatGPT 4.1 Mini, **M2**: DeepSeek R1, **M3**: Llama 3.3 70B, **M4**: Haiku 3.5, **M5**: Sonnet 3.7, **M6**: Sonnet 3.7 (Think Mode).

H Satisfaction Condition Validation Details

We conducted a targeted evaluation to assess the accuracy and completeness of automatically extracted user satisfaction conditions. We randomly sampled 10 GitHub issues per language across seven programming languages and collected annotations from three independent human raters. Each annotator evaluated two aspects: **Condition Accuracy** and **Condition Completeness**. Across 210 records, annotators reviewed a total of 663 satisfaction conditions. The detailed per-language breakdown is shown in Table 7.

Annotator Agreement. We computed pairwise agreement and Cohen's Kappa across the three annotators. The average percentage overlap was 82.39%, with some pairs reaching perfect agreement (100.00%, $\kappa = 1.0$) and others showing low or negative κ, highlighting variability in interpretation and the need for further calibration in future rounds.

https://doi.org/10.52202/085713-4280

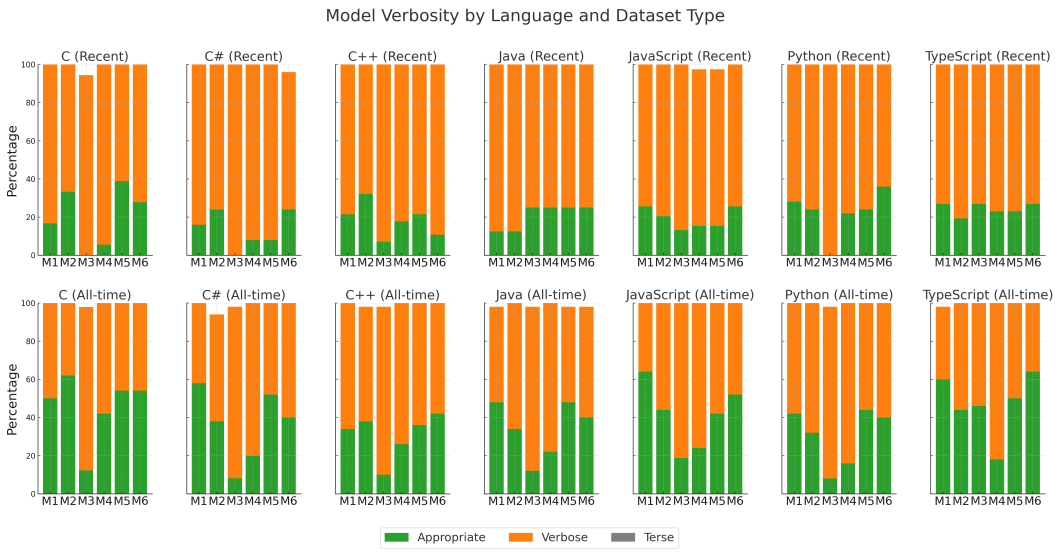

Figure 7: Cumulative verbosity distribution of model responses across seven programming languages. Each bar shows the proportion of responses classified as **Appropriate** (green), **Verbose** (orange), and **Terse** (gray, rarely observed). The top row corresponds to evaluations on **recent repositories**, while the bottom row shows results on **all-time repositories**. Models are labeled as M1–M6 in the following order: **M1**: ChatGPT 4.1 Mini, **M2**: DeepSeek R1, **M3**: Llama 3.3 70B, **M4**: Haiku 3.5, **M5**: Sonnet 3.7, **M6**: Sonnet 3.7 (Think Mode). Most models tend to generate verbose responses, with only a few models (notably M6) achieving higher rates of appropriately concise answers.

Table 7: Human Alignment Evaluation: Accuracy, Recall, and Additional Condition Needs per Language

Language	Total Conditions	Correct	Incorrect	Accuracy	Recall	Additional Needed (%)
C	87	80	7	91.95%	96.67%	3.3%
C++	96	94	2	97.92%	26.67%	73.3%
C#	96	68	28	70.83%	70.00%	30.0%
Java	102	87	15	85.29%	86.67%	13.3%
JavaScript	93	80	13	86.02%	53.33%	46.7%
Python	102	79	23	77.45%	66.67%	33.3%
TypeScript	87	84	3	96.55%	60.00%	40.0%
Overall	663	572	91	86.27%	65.71%	34.3%

I Human Annotation Instructions

This section provides the full instructions shown to human annotators for evaluating the quality of user satisfaction conditions in CodeAssistBench (CAB). These instructions were designed to ensure consistency, transparency, and alignment with user intent during the evaluation process.

Ethics and Compensation

Annotators were external contractors who provided informed consent prior to participation. They were compensated at or above the local minimum wage. The task involved minimal risk, and no personally identifiable or sensitive information was collected.

Task Overview

This annotation task involves evaluating user satisfaction conditions automatically extracted from GitHub issue threads. A satisfaction condition describes **what** the user needs from the main-

tainer's response—not **how** to implement it. These conditions should reflect the user's goals, constraints, or expectations based on the full conversation.

Each annotator was asked to review 70 GitHub issues, with three annotators assigned per issue. A calibration phase with 10 examples was used to ensure alignment before the full task began.

Annotation Interface

Annotators were presented with the following information for each task:

- Issue title and initial question
- Full GitHub thread, including all comments and author roles
- The list of model-generated satisfaction conditions

Annotator Instructions (Verbatim)

The following is the exact text shown to annotators:

> **Overview**
> This project focuses on verifying user satisfaction conditions automatically extracted from GitHub issue conversations using a language model. A satisfaction condition describes what the user needs from the maintainer's response—not how it is implemented.
>
> You will review GitHub issues and evaluate each proposed satisfaction condition for: - **Correctness**: Does the condition accurately reflect user needs? - **Completeness**: Are there missing conditions that should be included?
>
> **For each issue, you will be given:** - The original GitHub title and user question - All follow-up comments (including roles and timestamps) - The model-generated satisfaction conditions
>
> **Your tasks:** 1. For each listed condition: - Mark it as `Correct` or `Incorrect` - Provide a short justification (e.g., "User confirmed this need in a follow-up comment.") 2. Suggest any **missing conditions** that should be added, with justification.
>
> **Annotation Format:**

```
{
  "original_conditions": [
    {
      "condition": "string",
      "judgment": "Correct" | "Incorrect",
      "justification": "string"
    }
  ],
  "additional_conditions (if any)": [
    {
      "condition": "string",
      "justification": "string"
    }
  ]
}
```

> **Levels of Abstraction for Conditions:**
>
> Avoid overly specific or overly generic phrasing. Good conditions should reflect the *what*, not the *how*.
>
> Too Specific (Avoid): - "Use numpy.where(condition, x, y)" - "Set max_depth=5 in the RandomForest constructor"
>
> Good Abstraction Level: - "A vectorized approach to conditional element selection" - "Guidance on suitable tree depth settings"
>
> Too Generic (Avoid): - "A working solution" - "Help with the problem"

 https://doi.org/10.52202/085713-4280

Criteria for a Good Condition: - **Transferable**: Not tied to a specific implementation - **Verifiable**: Can be clearly judged as satisfied or not - **Evidenced**: Grounded in what the user said or implied - **Needs-Focused**: Describes the user's goal/problem—not the method

Example Annotation

Below is an anonymized real example used during calibration:

Issue Title: "How can I play RTSP stream without audio codecs?"

User Post: *How can I play RTSP stream without audio codecs? I need only video. I can't start watching because the camera uses G.711 for audio.*

Model-Generated Conditions:

- Explanation of how the player handles unsupported audio codecs
- Confirmation that video playback is possible without audio codec support

Annotator Output:


```
{
  "original_conditions": [
    {
      "condition": "Explanation of how the player handles unsupported audio codecs",
      "judgment": "Correct",
      "justification": "User asked about G.711 and the maintainer explained it would be
     dropped automatically."
    },
    {
      "condition": "Confirmation that video playback is possible without audio codec support",
      "judgment": "Correct",
      "justification": "The maintainer confirmed that video-only playback is supported."
    }
  ],
  "additional_conditions": []
}
```


Quality Control

Each annotator completed a 10-example calibration round. Only those with consistent agreement advanced to the full task. Inter-annotator agreement statistics are reported in Appendix H.

Annotator Backgrounds

A total of 21 annotators participated in the human evaluation task. All annotators had at least 2 years of programming experience (mean: 5.1 years), with backgrounds spanning industry roles in software engineering, quality assurance, and backend development.

- **Languages covered:** JavaScript/TypeScript, Python, Java, C#
- **Common frameworks:** React, Angular, Spring Boot, Django, .NET Core
- **Cloud & infra skills:** AWS, SQL, HTML/CSS
- **Review process:** Each GitHub issue was annotated by 3 independent reviewers.

A full table of individual annotator experience is available upon request but omitted from the appendix to preserve readability.

J Synthetic Dataset Ablation Study

To investigate whether the observed performance gap between recent and all-time repositories stems from AI-generated code characteristics versus training data knowledge limitations, we conducted an ablation study using a synthetic dataset.

J.1 Methodology

We randomly selected 50 problems from the recent repository subset and generated fully synthetic reproductions using a maintainer agent backed by Claude Sonnet 3.7. For each issue, the agent was run iteratively until it confirmed the issue was fully reproducible within the generated codebase. Each synthetic repository maintained similar complexity (20-28 files, 1,500+ lines of code) while preserving the identical bugs and satisfaction conditions from the original issues. The synthetic repositories were designed to use programming language versions and library versions that fall within the model's training knowledge cutoff.

We initially hypothesized that AI-generated code characteristics might be a primary contributor to the observed low accuracy on recent repositories. However, during repository generation, the maintainer agent naturally produced extensive documentation as part of its workflow to ensure full reproducibility, resulting in synthetic repositories with significantly better documentation coverage than typical real-world repositories.

J.2 Results

The synthetic dataset evaluation revealed a striking finding: Claude 3.7 Sonnet achieved **74% accuracy** on the synthetic repositories, substantially higher than the 11.34% accuracy observed on the recent repository subset (Table 2).

Table 8: Performance comparison across repository types (Sonnet 3.7)

Dataset Type	Correct	Partial	Incorrect	Total Accuracy
All-time repos	25.71%	35.43%	38.86%	61.14%
Recent repos	11.34%	30.93%	57.73%	42.27%
Synthetic repos	**74.0%**	16.0%	10.0%	**90.0%**

J.3 Analysis and Implications

The superior performance on synthetic code compared to recent repositories provides compelling evidence that **the performance gap is primarily driven by training data limitations rather than inherent properties of AI-generated code**. Several key factors explain this finding:

Library Version Knowledge The most significant factor contributing to failures in recent repositories was the use of library versions and programming language features released after the model's training data cutoff. Recent repositories frequently employ:

- Latest library APIs introduced after the training cutoff
- Modern language features from recent standard updates
- Newly released frameworks and their evolving best practices

In contrast, the synthetic repositories were constructed using language and library versions that fall well within the model's training knowledge, enabling the model to leverage its full understanding of these technologies.

Repository Complexity Synthetic repositories, while maintaining realistic complexity (1,500+ LOC, professional structure), were necessarily smaller and more focused than many production codebases in the recent repository set. The average repository size in the synthetic dataset was approximately 60% of the recent repository average, reducing the cognitive load required for code comprehension and bug localization.

Documentation and Code Quality The AI-generated synthetic repositories exhibited consistent documentation practices and code organization patterns familiar to the model, as they were generated using the model's own understanding of best practices. This consistency may have facilitated more effective code navigation and problem-solving.

 https://doi.org/10.52202/085713-4280

J.4 Implications for Benchmark Design

These findings have important implications for understanding model capabilities:

1. **Training Data Recency Matters**: The 62.66 percentage point difference between recent repositories (11.34%) and synthetic repositories (74.0%) primarily reflects knowledge cutoff limitations rather than fundamental reasoning deficits.
2. **True Reasoning Capability**: The 74% accuracy on synthetic repositories suggests that when provided with code using familiar language features and libraries, Claude 3.7 Sonnet demonstrates strong technical reasoning and problem-solving abilities.
3. **Repository Context Scaling**: The similarity between synthetic (74.0%) and all-time repository performance (25.71%) when using the same evaluation model suggests that both repository complexity and knowledge cutoff contribute to performance differences.

J.5 Limitations

This ablation study has several limitations that should be considered:

- The synthetic repositories, while realistic, may not fully capture the complexity and idiosyncrasies of production codebases
- AI-generated code may inadvertently conform to patterns more easily recognized by the generating model
- The sample size of 50 issues, while substantial, represents a subset of the recent repository dataset

J.6 Conclusion

The synthetic dataset ablation study demonstrates that the primary challenge in evaluating modern code repositories is not the nature of the code itself, but rather the rapid evolution of programming ecosystems beyond model training data. This finding suggests that regular model updates aligned with current software development practices would likely narrow the performance gap significantly. It also validates CodeAssistBench as an effective benchmark for measuring both current capabilities and the impact of training data recency on practical coding assistance performance.

K Error Category Analysis

To understand model strengths and weaknesses across different problem types, we performed automated error categorization on the 620 human-annotation records (310 agent responses $\times$ 2 annotators) from our expert validation study (Section B). This analysis clusters issues into coarse bug types, revealing targeted patterns in model capabilities.

K.1 Methodology

We used Claude Sonnet 3.7 to categorize each GitHub issue into one of seven error types based on the issue title, body, and conversation context. The categories are:

- **logic**: Algorithm bugs, incorrect behavior, unexpected results, calculation errors
- **environment**: Configuration, setup, Docker, build systems, environment variables, paths
- **dependency**: Package/module imports, version conflicts, installation issues
- **api**: REST/GraphQL endpoints, HTTP requests, authentication, API integration
- **syntax**: Parse errors, type errors, undefined references, compilation errors
- **performance**: Speed, memory, optimization, timeout issues
- **other**: Documentation, feature requests, questions that don't fit technical error categories

For each issue, the LLM provided: (1) the primary category, (2) confidence level (high/medium/low), and (3) a brief reasoning explaining the classification. Categorizations were cached to ensure

reproducibility. The vast majority (97%+) of classifications were assigned high confidence, indicating reliable categorization.

K.2 Category Distribution

Table 9 shows the distribution of issues across error categories. Logic errors (28.8%) and environment configuration issues (28.5%) dominate the dataset, together accounting for over half of all issues. This reflects the reality that real-world programming assistance often involves debugging algorithmic problems and resolving setup/configuration challenges. Dependency issues comprise 14.6% of the dataset, while API integration, syntax errors, and performance problems occur less frequently.

Table 9: Distribution of error categories across 620 annotation records

Category	Count	Percentage
Logic	178	28.8%
Environment	176	28.5%
Other	104	16.8%
Dependency	90	14.6%
API	40	6.5%
Performance	20	3.2%
Syntax	12	1.9%
Total	620	100.0%

K.3 Model Performance by Category

Table 10 presents LLM judge agreement rates with human annotators across different error categories. The analysis reveals significant variation in model capabilities depending on problem type.

Table 10: LLM judge agreement rates by error category across three evaluation metrics

Category	Technical Correctness	Verbosity	Verdict
Performance	75.00%	75.00%	75.00%
Syntax	58.33%	75.00%	75.00%
Logic	60.11%	79.78%	62.36%
Environment	59.66%	75.57%	60.23%
API	60.00%	65.00%	62.50%
Other	58.65%	78.85%	61.54%
Dependency	55.56%	77.78%	55.56%

K.4 Key Findings

Our error category analysis reveals several important patterns:

Performance Issues Show Highest Agreement. Models achieve 75% agreement with human evaluators on performance-related issues (OOM errors, timeouts, memory leaks). This suggests that models are particularly effective at diagnosing resource constraints and performance bottlenecks, likely because these issues have clear symptoms and well-established solutions.

Dependency Resolution Remains Challenging. With only 55.6% agreement on both technical correctness and verdict, dependency-related issues (package conflicts, version mismatches, missing modules) represent the most difficult category for models. This aligns with real-world developer pain points, where resolving complex dependency graphs often requires deep ecosystem knowledge and version compatibility awareness.

Logic and Environment Issues Dominate. The prevalence of logic errors (28.8%) and environment issues (28.5%) in our dataset reflects the reality of programming assistance - developers

 https://doi.org/10.52202/085713-4280

frequently need help debugging algorithms and configuring development environments. Models achieve moderate performance on these categories (59-60% technical correctness), indicating room for improvement on the most common real-world assistance scenarios.

Syntax Errors Are Rare But Well-Handled. Only 1.9% of issues involve syntax errors. However, models show strong performance when they occur (75% verdict agreement), suggesting reliable handling of structural code errors.

These findings provide actionable insights for improving code assistance systems: prioritizing better dependency resolution capabilities, enhancing environment configuration support, and maintaining strong performance on the logic debugging tasks that developers encounter most frequently.

L API Testing and Specification-Based Evaluation

While CodeAssistBench focuses on conversational programming assistance, complementary work in automated API testing demonstrates the value of specification-grounded evaluation that shares conceptual similarities with our satisfaction condition approach. Recent research has explored automated testing for REST APIs using OpenAPI specifications as structured artifacts Kim et al. [2022, 2023], leveraging NLP techniques to extract semantic information including parameter constraints and endpoint dependencies. Subsequent work has demonstrated how large language models can improve REST API testing by generating semantically meaningful test inputs Kim et al. [2024], with multi-agent systems further enhancing test effectiveness by modeling complex API interactions Kim et al. [2025a]. More recent work explores whether small language models can effectively handle domain-specific testing tasks when provided with clear specifications Kim et al. [2025b], demonstrating that specification quality significantly impacts model effectiveness—a finding that aligns with our observation that documentation quality affects resolution success in CAB. While API testing benchmarks target functional correctness through executable specifications (OpenAPI) and CAB evaluates conversational assistance through extracted satisfaction conditions, both approaches share a fundamental principle: **specification-grounded evaluation** that moves beyond subjective assessment toward principled evaluation based on explicitly stated requirements, whether derived from formal specifications or real developer conversations.

Building on these insights, we plan to extend CAB to include web API assistance scenarios. This extension will leverage OpenAPI specifications from popular API repositories to automatically generate programming assistance questions about API integration, authentication, error handling, and endpoint usage. Similar to our GitHub issue pipeline, we will extract satisfaction conditions from developer forums and Stack Overflow discussions related to specific APIs, combine them with their OpenAPI specifications to create grounded evaluation scenarios, and simulate multi-turn conversations where models must help developers integrate and troubleshoot API usage. This specification-grounded approach will enable systematic evaluation of API assistance capabilities while maintaining the conversational, multi-turn nature that distinguishes CAB from traditional functional testing benchmarks.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The Abstract and Section 1 clearly articulate CAB's three core contributions: (1) an automated dataset generation pipeline, (2) a realistic multi-turn Q&A dataset grounded in GitHub issues, and (3) a chat-driven evaluation framework. These claims are consistently supported by the methods and experiments presented throughout the paper, ensuring alignment between stated contributions and actual work.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We discuss the limitations in Section 5, including issues such as conservative satisfaction condition extraction (favoring precision over recall), limited evaluation coverage due to sampling, and simulation artifacts from templated user behavior. These limitations help contextualize the scope and applicability of our benchmark and findings.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important

 https://doi.org/10.52202/085713-4280

role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Justification: This work is entirely empirical and does not present formal theorems or proofs.

   Guidelines:

   - The answer NA means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We detail the dataset generation pipeline, evaluation setup, and sampling strategy in Section 3 and Section 4. All code, data, prompts, and instructions needed to reproduce the main experimental results are publicly available at the URL provided in the Abstract.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: As stated in the Abstract, the complete CAB benchmark, evaluation framework, and dataset are available at `https://anonymous.4open.science/r/CAB-CBA3/`. The repository includes instructions for accessing the raw and processed data, build environments, and running the evaluation pipeline.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: We describe the evaluation subset construction—including sampling by language, turn length, and Docker presence—in Section 4. Model generation parameters, including fixed temperatures (0 for dialogue and 0.7 for Dockerfile generation), are specified in Appendix C. The full evaluation setup, prompts, and execution logs are available at `https://anonymous.4open.science/r/CAB-CBA3/`.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

https://doi.org/10.52202/085713-4280

Justification: Due to the multi-turn, agentic nature of CAB's evaluation—where each model response triggers follow-up interactions and environment-level validation—each data point is expensive to generate. Running multiple trials per issue (e.g., with varied sampling or decoding) was not feasible at scale. Nevertheless, the evaluation spans over 500 diverse issues across 7 languages and multiple turn lengths, offering reliable performance trends despite the lack of formal confidence intervals.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The paper provides compute details in Appendix E, including hardware specifications, resource requirements, and an estimate of total compute usage.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We use only publicly available GitHub issues and apply LLM-based filtering for personally identifiable information (PII), as described in Appendix A.1, to ensure that no sensitive personal data is included in our dataset or analysis.

   Guidelines:

- The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: We include a brief note in the Conclusion directing readers to Appendix D, where we discuss CAB's potential societal benefits—such as improving developer productivity and transparency in LLM-based tools—and risks, including overreliance on automated suggestions and possible misuse in security-sensitive environments.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: This work does not release any high-risk assets such as generative models or sensitive datasets; thus, safeguards are not applicable.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.

https://doi.org/10.52202/085713-4280

- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: In Section 3.1.1, we restrict repository selection to those under permissive licenses, including MIT, Apache-2.0, ISC, Zlib, CC-BY, and CC0-1.0, to ensure license compliance. Original repository authors are credited in our metadata, which includes license names and source URLs.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The CAB dataset's structure and components are described in Section 3. Documentation including usage instructions, license, and known limitations is provided at `https://anonymous.4open.science/r/CAB-CBA3/`.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [Yes]

    Justification: The paper includes a complete description of the annotation task and protocol in Appendix I, including the full text of instructions given to annotators, example inputs

and outputs, abstraction guidelines, and JSON formatting schema. Annotators were external contractors who provided informed consent and were compensated at or above the local minimum wage, in accordance with the NeurIPS Code of Ethics.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [Yes]

    Justification: The human annotation study was reviewed and approved through our organization's internal human subjects review process. Annotators were external contractors who provided informed consent prior to participation.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: We explicitly detail the use of Sonnet 3.7 for message filtering, environment generation, and satisfaction condition extraction in Sections 3.

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.

https://doi.org/10.52202/085713-4280