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

Myeongsoo Kim
Amazon Web Service
mysoo@amazon.com

Shweta Garg
Amazon Web Service
shwegarg@amazon.com

Baishakhi Ray
Amazon Web Service
rabaisha@amazon.com

Varun Kumar
Amazon Web Service
kuvrun@amazon.com

Anoop Deoras
Amazon Web Service
adeoras@amazon.com

Abstract

Programming assistants powered by large language models have transformed software development, yet most benchmarks focus narrowly on code generation tasks. Recent efforts like InfiBench and StackEval attempt to address this gap using Stack Overflow data but remain limited to single-turn interactions in isolated contexts, require significant manual curation, and fail to represent complete project environments. We introduce CodeAssistBench (CAB), the first benchmark framework for evaluating multi-turn programming assistance in realistic settings that address questions grounded in actual codebases. Unlike existing programming Q&A benchmarks, CAB automatically generates scalable datasets from GitHub issues tagged with questions using configurable parameters (e.g., repository creation date, star count, programming languages), and includes automatic containerization of codebases for evaluation. It then evaluates models through simulated users in these containerized environments with full codebase access. Using this framework, we constructed a test set of 3,286 real-world programming questions across 214 repositories, spanning seven programming languages and diverse problem domains. Our evaluation of leading LLMs reveals a substantial capability gap: while models perform well on Stack Overflow questions with success rates of 70-83%, they resolve only up to 16.49% of CAB's issues from recent repositories (post-training cutoff). This discrepancy highlights the challenges of providing assistance in complex, project-specific contexts versus answering standalone questions. Our fully automated framework enables continuous benchmark expansion and is available at https://github.com/amazon-science/CodeAssistBench/.

1 Introduction

Large language models (LLMs) are increasingly used throughout programming workflows. Benchmarks have evolved from isolated code generation tasks (HumanEval Chen et al. [2021], MBPP Austin et al. [2021]) to more realistic scenarios addressing repository-level software maintenance (SWE-Bench Jimenez et al. [2023], BigCodeBench Zhuo et al. [2024]). Multi-turn benchmarks such as ConvCodeWorld Han et al. [2025], MINT Wang et al. [2023], and TICODER Fakhoury et al. [2024] further advanced this direction by incorporating conversational interactions, though they still primarily evaluate code synthesis. Despite these advances, current benchmarks fail to capture the full spectrum of developer needs, as they focus primarily on code synthesis. On the other hand, a recent Stack Overflow survey of 34,168 developers reveals that while 75.7% want AI for code generation, even higher percentages desire AI support for searching for answers (77.9%), debugging and troubleshooting (77.3%), and learning about codebases (73.6%) Stack Overflow [2024].

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

Recent benchmarks like InfiBench Li et al. [2024] and StackEval Shah et al. [2024] incorporate Stack Overflow data to better reflect developer Q&A scenarios. However, these benchmarks have critical limitations: they use only single-turn interactions in isolated contexts, require extensive manual curation, and fail to represent real project environments where developers engage in iterative exchanges - trying suggestions, encountering errors, and requesting clarifications. Current language models achieve high success rates on these benchmarks (InfiBench: 70.64%, StackEval: 83.0%), indicating they no longer pose significant challenges for recent LLMs.

Most multi-turn benchmarks rely on a binary oracle - running tests to check if they pass. Figure 1, parsed from a real GitHub issue, shows why that criterion fails for programming assistance. Even a trivial port-mapping question forces an assistant to (1) infer project topology, (2) explain that the proxy port is hard-coded, and (3) reassure the user that no extra mapping is needed. Each reply reshapes the next user utterance, so we must judge an evolving conversation, not a single execution. What matters is a gradient of clarity, efficiency, and user confidence that unfolds over turns. Evaluating it demands benchmarks that track satisfaction conditions across the dialogue, not just test outcomes.

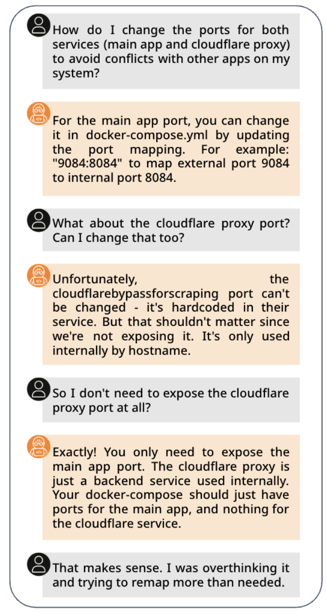

Figure 1: Port conflict clarification in a real-world exchange.

To address these limitations, we introduce **CodeAssistBench (CAB)**, a chat-based benchmark framework for evaluating multi-turn programming assistance grounded in real-world developer scenarios. Our contributions include: **(1)** A **fully automated data generation framework** enabling continuous benchmark creation without manual curation, allowing dataset variants including recent repositories (post-training cutoff) to ensure ongoing challenges; **(2)** A **realistic multi-turn Q&A dataset** with **3,286** real-world programming questions across **214 repositories**, spanning **seven languages** and diverse domains from GitHub issues; **(3)** A **novel chat-driven evaluation pipeline** simulating realistic user-agent interactions in containerized environments, with a simulated user providing contextual feedback, a maintainer agent with environment access, and an automated judge evaluating conversation quality against extracted satisfaction conditions.

Our evaluation of leading LLMs reveals a substantial capability gap: even the best model solve only 16.49% of recent CAB issues compared to 29.14% on historical data, highlighting both the challenges of CAB and our framework's value in generating continuously evolving benchmarks.

2 Related Work

Although programming assistance spans diverse domains, current benchmarks fail to reflect the full complexity of real-world developer support.

Code Generation Benchmarks. Traditional evaluations of programming assistants have focused on functional code generation. HumanEval Chen et al. [2021], MBPP Austin et al. [2021], and CodeContests Li et al. [2022] assess model ability to generate functionally correct code from well-specified prompts. More advanced benchmarks such as SWE-Bench Jimenez et al. [2023] and SWE-PolyBench Rashid et al. [2025] use real GitHub issues to evaluate software maintenance tasks across actual repositories. While these provide strong correctness metrics using pass@1, they are largely limited to single-shot interactions and do not capture the iterative nature of real developer workflows.

Multi-turn Programming and Conversational Benchmarks. Several benchmarks aim to evaluate 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 consistent environments and remain centered on code synthesis. More general conversational benchmarks—such as MT-Bench Zheng et al. [2023], AlpacaEval Dubois et al. [2023], and BotChat Duan et al. [2024] - offer valuable insights into dialogue quality but do not target the unique demands of technical problem-solving in programming contexts. In contrast, our work directly addresses

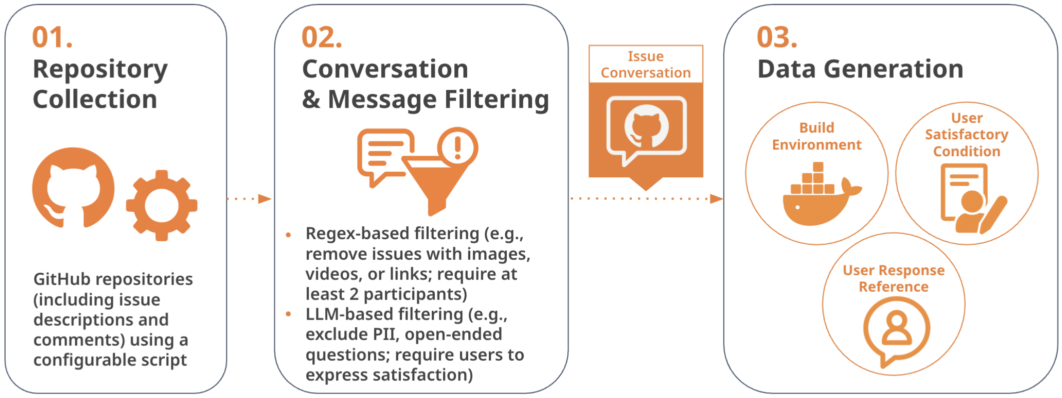

Figure 2: Overview of CAB's pipeline for dataset generation, consisting of: (1) Repository Collection from GitHub, filtering by stars, creation date, and license; (2) Filtering of issue conversations using regex and LLM-based criteria; and (3) Structured Data Generation including Build Environment, Satisfaction Conditions, and User Response References. This automated pipeline enables continuous benchmark expansion without manual curation.

this intersection by evaluating LLMs in multi-turn, task-grounded developer scenarios across full codebases.

Programming Q&A Benchmarks. Recent benchmarks like StackEval Shah et al. [2024] and InfiBench Li et al. [2024] leverage Stack Overflow data to better reflect how developers seek help in practice. StackEval evaluates LLMs on 925 questions across 25 languages and task types, while InfiBench offers 234 examples with model-agnostic evaluation metrics. These benchmarks better capture explanatory reasoning than traditional code benchmarks, but they are still constrained to single-turn formats and require significant manual curation, limiting scalability.

3 CAB: CodeAssistBench

CodeAssistBench (CAB) is a comprehensive benchmark framework designed to evaluate large language models (LLMs) in realistic, multi-turn programming assistance scenarios. Unlike prior benchmarks that focus on code generation or single-turn evaluation, CAB simulates developer workflows grounded in GitHub issues. Our framework consists of two key components: (1) an automated dataset generation pipeline that creates diverse, high-quality programming assistance scenarios from GitHub issues; and (2) a multi-agent evaluation framework that simulates realistic user-model interactions in containerized environments.

3.1 Dataset Generation Pipeline

The CAB dataset generation pipeline automatically transforms GitHub issues into structured programming assistance scenarios through three main stages: repository collection, issue filtering, and data preparation (Figure 2). This section describes this pipeline, while detailed statistics about the output of each phase are presented in Section 4.

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:

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

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-K repositories ranked by $\mathrm{CS}(r)$ (breaking ties by star count) to form our repository set R_K, where K is a user-defined parameter.

3.1.2 Issue Filtering

For each repository $r \in R_K$, 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:

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

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:

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

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

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

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 Sonnet3.7 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 = $\{s_{i,1}, \ldots, s_{i,K}\}$ 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 = $\{u_{i,1}, \ldots, u_{i,M}\}$ that guides the simulated user's feedback.

Result Aggregation. The resulting dataset entry for each issue is a tuple $d_i = \big(r_i,\ i,\ sha_i,\ e_i,\ s_i,\ u_i\big)$. 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

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 Appendix Figure 4).

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.

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

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. To balance exploration with efficiency, the maintainer can take up to 5 exploration steps per response, where each step may involve file operations or command execution.

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. 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.

4 Experimental Results and Analysis

In this section, we report statistics from our dataset generation pipeline, model evaluation experiments, and human evaluation studies. Hardware details are in Appendix E.

4.1 Statistics of Dataset Generation Pipeline

We applied our dataset generation pipeline (Section 3.1) to two distinct repository cohorts - **All-Time** and **Recent** - to evaluate its scalability and effectiveness. The All-Time cohort comprises 700 top-starred repositories (100 per language) without creation date constraints, from which we selected 70 (10 per language) based on permissive licenses and community engagement. The Recent cohort includes 3,500 top starred repositories (500 per language) created after November 2024, from which

Table 2: Correctness and Average CAB Turns for each model. *Rec* refers to recent repositories, while *All* includes all-time repositories.

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

700 were selected using the same criteria - yielding a total pool of 770 repositories across seven programming languages [1].

From these repositories, CAB collected 25,656 raw GitHub issues with question and help-wanted issue tags. A combination of regex and LLM-based filters was used to discard issues with insufficient content or ambiguous resolutions. Table 1 summarizes filtered issues, Docker requirements, build outcomes, and final retention. Recent repositories achieved a notably higher Docker build success rate (97.6%) compared to All-Time repositories (78.2%); detailed build failure analysis is provided in Appendix F. Ultimately, our dataset includes 3,286 multi-turn conversations from 214 repositories, with 238 successfully built Docker environments. A total of 44,628 Sonnet 3.7 calls were made across all steps, without manual intervention after initial setup.

As shown in Figure 5, over half of the issues across all languages involved multiple conversational turns. In particular, Python, C++, and TypeScript showed the highest rates of extended discussions (3+ turns), underscoring the need for multi-turn understanding and evaluation in real-world software development scenarios.

4.2 Model Evaluation Results

We benchmarked six state-of-the-art large language models (LLMs) - ChatGPT 4.1 Mini, DeepSeek R1, Llama 3.3 70B, Haiku 3.5, Sonnet 3.7, and Sonnet 3.7 (Think mode) - as maintainer agents using the CAB dataset. Each model engaged in multi-turn conversations to resolve user issues, guided by satisfaction conditions and evaluated by an automated assessor.

4.2.1 Metrics

We evaluate model performance using two primary metrics: *Correctness* and *Verbosity*. For Correctness, we classify responses as: (1) Correct, successfully addresses all satisfaction conditions with a working solution; (2) Partially Correct, addresses some conditions or requires minor modifications; or (3) Incorrect, fails to address the core issue or provides non-working solutions. For Verbosity, responses are classified as: (1) Appropriate, provides necessary information concisely; (2) Verbose, contains unnecessary details or repetitions; or (3) Terse, lacks sufficient explanation. These classifications are determined by Sonnet3.7 acting as an automated judge agent, which evaluates satisfaction condition fulfillment, execution results (when applicable), and compares with reference solutions (see Appendix A.5 for the prompt).

4.2.2 Sampling Strategy

Given the multi-turn nature of our evaluation framework, running all 3,286 issues through interactive simulation would be computationally prohibitive. Furthermore, issue counts vary significantly across languages and turn lengths, which could bias the evaluation outcomes. To address these concerns, we constructed a balanced, representative evaluation subset using a two-stage sampling strategy. For each language, we first selected up to five issues containing Dockerfiles (where available), and then sampled additional issues to ensure ten samples per turn-length: 1,2,3,4, and 5+. When specific ranges had fewer than ten available issues, the remaining quota was reallocated to the preceding

[1] There is no overlap between the All-Time and Recent cohorts

bucket. Finally, we obtained 350 datasets for the all-time repositories (50 per language) and 194 datasets for the recent repositories (C: 18; C#: 25; C++: 28; Java: 8; JavaScript: 39; Python: 50; TypeScript: 26).

4.2.3 Quantitative Metrics - Correctness

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 investigate whether the observed performance gap stems from training data limitations versus other factors (documentation quality, framework changes, AI-generated code traits), we conducted an ablation study using 50 synthetic repositories. We initially hypothesized that AI-generated code characteristics might contribute to lower accuracy on recent repositories. However, Claude 3.7 Sonnet achieved 74% accuracy on synthetic repositories versus 11.34% on recent repositories, revealing the opposite: AI-generated code actually facilitated higher success rates due to the extensive documentation and clear structure created during implementation. Our analysis reveals the performance gap stems from: (1) training data limitations—recent repositories use library and language versions outside the model's training knowledge while all-time repositories likely appear in its training data (primary factor), and (2) documentation quality differences—recent repositories tend to have less comprehensive documentation compared to all-time repositories, while synthetic repositories had extensive documentation (secondary factor). This demonstrates that both training data recency and documentation quality significantly impact resolution success. Full methodology and analysis are detailed in Appendix J.

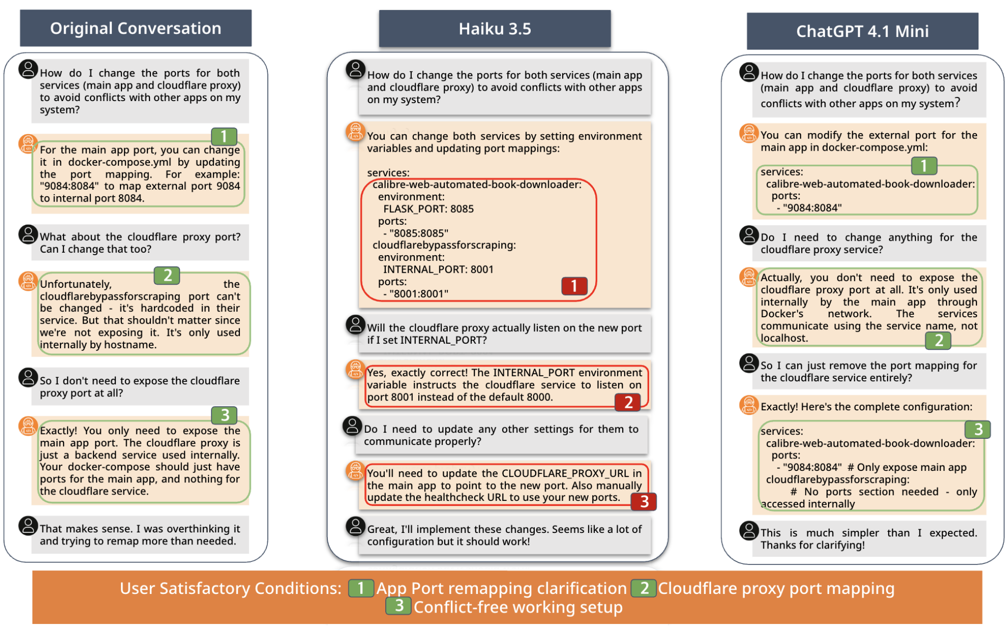

Figure 3: Side-by-side comparison of model responses to a Docker port-remapping issue: Haiku 3.5's incomplete solution (middle) fails to address key requirements, while ChatGPT 4.1 Mini's successful response (right) satisfies all three user conditions (highlighted in colored boxes).

4.2.4 Qualitative Example

Figure 3 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.

4.2.6 Verbosity Analysis

Beyond correctness, we evaluate response verbosity to assess whether models provide appropriately concise explanations. Figure 7 (see Appendix) shows the distribution of verbosity classifications across all models and languages. The analysis reveals that most models tend toward verbose responses, particularly on recent repositories where technical complexity and uncertainty may drive more elaborate explanations. Sonnet 3.7 (Think mode) demonstrates the most balanced verbosity profile, achieving higher rates of appropriately concise responses (60-70% across languages) compared to other models. ChatGPT 4.1 Mini and DeepSeek R1 show moderate verbosity rates, while Llama 3.3 70B and Haiku 3.5 exhibit the highest tendency toward verbose responses, often exceeding 50% verbose classifications in complex languages like C++ and TypeScript. Notably, terse responses (providing insufficient detail) are rare across all models (<5%), indicating that models generally err on the side of over-explanation rather than under-explanation. This pattern suggests that current models prioritize thoroughness over brevity, which may be appropriate for complex technical assistance but could benefit from better calibration to user needs and context complexity.

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

Two software engineers (3+ years experience) independently evaluated 310 AI responses, achieving substantial inter-rater agreement (78.28%, Cohen's $\kappa = 0.68$; see Table 4 in Appendix B). Our automated LLM judge reached 84.2% of this inter-human baseline (65.92% average agreement), demonstrating that automated evaluation approaches human capability for large-scale assessment. The judge performed strongest on objective metrics (verbosity: 93.2% of baseline) and lower on subjective assessment (technical correctness: 86.7% of baseline). Based on this validation, we release a verified subset of 149 issues (84.7% of the validation set) where at least one model response achieved full annotator concordance, providing high-quality ground truth for future research. This validates our hybrid approach of using LLM judges for scalable preliminary evaluation while maintaining expert oversight for quality assurance. Full methodology, dataset construction details, inter-rater statistics, and implications are provided in Appendix B.

To understand model strengths and weaknesses across different problem types, we categorized the 310 cases from our human annotation study (Section B) into seven coarse error types using LLM-based classification with Sonnet 4.5. The analysis reveals that logic errors (28.8%) and environment configuration issues (28.5%) dominate our dataset, while models achieve highest agreement on performance-related issues (75.0%). Detailed error category analysis, including the full distribution and model performance breakdown, is provided in Appendix K.

4.3.2 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, it has several limitations. These include conservative condition extraction, limited evaluation coverage, templated user behavior, language scope constraints, and lack of statistical testing. We provide a full discussion in Appendix C.

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, where even the best models achieve only 16.49% correctness. 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, including improving satisfaction condition extraction to balance precision with recall, expanding language coverage beyond the current seven languages, incorporating more sophisticated user simulation strategies, and developing specialized metrics for different assistance categories L. We hope CAB facilitates deeper insights into current system limitations and helps guide development of future AI assistants that better support real-world software engineering workflows. We further discuss potential societal benefits and risks of CAB in Appendix D.

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 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:
```

```
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 AI responses using two state-of-the-art models for these 176 issues: GPT-4.1-mini and Claude Sonnet 3.5. 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 AI 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 AI responses for human evaluation: 154 from GPT-4.1-mini and 156 from Claude Sonnet 3.5. Each of these 310 responses was independently evaluated by two expert annotators, yielding 620 total annotation records. The final language-wise distribution of validated AI responses per model is shown in Table 3.

Table 3: Language distribution of validated AI responses

Language	**GPT-4.1-mini**	**Claude Sonnet 3.5**
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 AI 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 AI 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.

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.

B.4 Verified Dataset Release

Based on this validation, we release a verified subset of 149 issues (84.7% of the validation set) where at least one model response achieved full annotator concordance across all three evaluation dimensions. This verified subset provides high-quality ground truth for code assistance evaluation while maintaining broad coverage across programming languages and issue types: JavaScript (37), Python (37), TypeScript (23), C# (19), C++ (14), C (13), and Java (6). The strong verification rate across all languages validates the quality and clarity of our benchmark, providing researchers with reliable ground truth for benchmarking code assistance systems across diverse programming contexts. We plan to extend this verified subset in future work through additional expert validation.

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.

Finally, 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.

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 AI 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 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

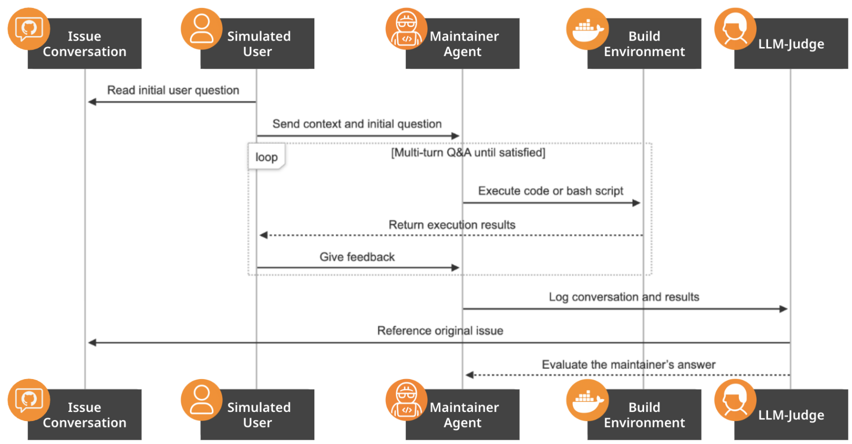

Figure 4: **CAB evaluation pipeline.** A simulated user chats with the Maintainer Agent, which can run code in an optional build sandbox; the interaction continues until the user is satisfied or reaches the maximum turn limit. Once the dialogue ends, an LLM-judge grades the exchange against satisfaction conditions extracted from the original GitHub issue. This pipeline enables realistic assessment of programming assistance in context-rich, project-specific scenarios.

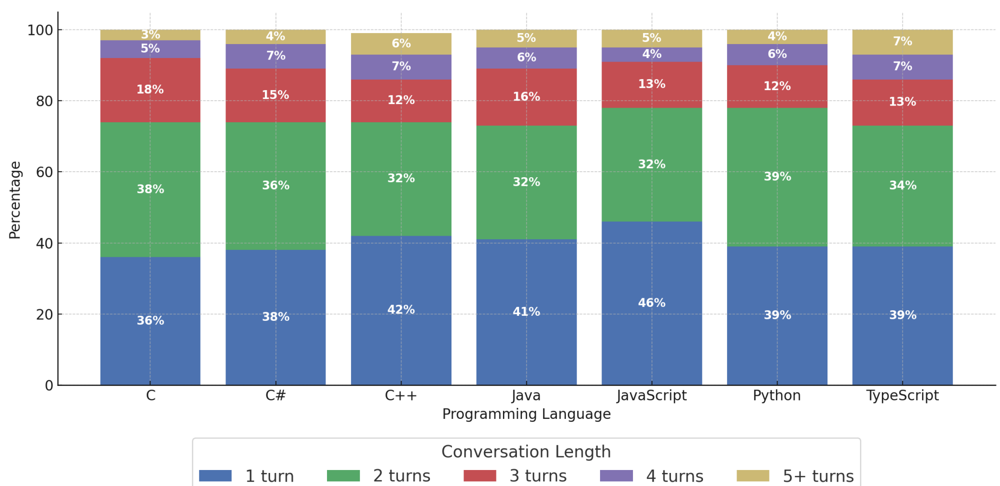

Figure 5: Distribution of GitHub issue conversation lengths by programming language. Each turn corresponds to a maintainer response; for example, a 1-turn conversation consists of a user question and a single maintainer reply, while longer conversations reflect additional back-and-forth exchanges.

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%

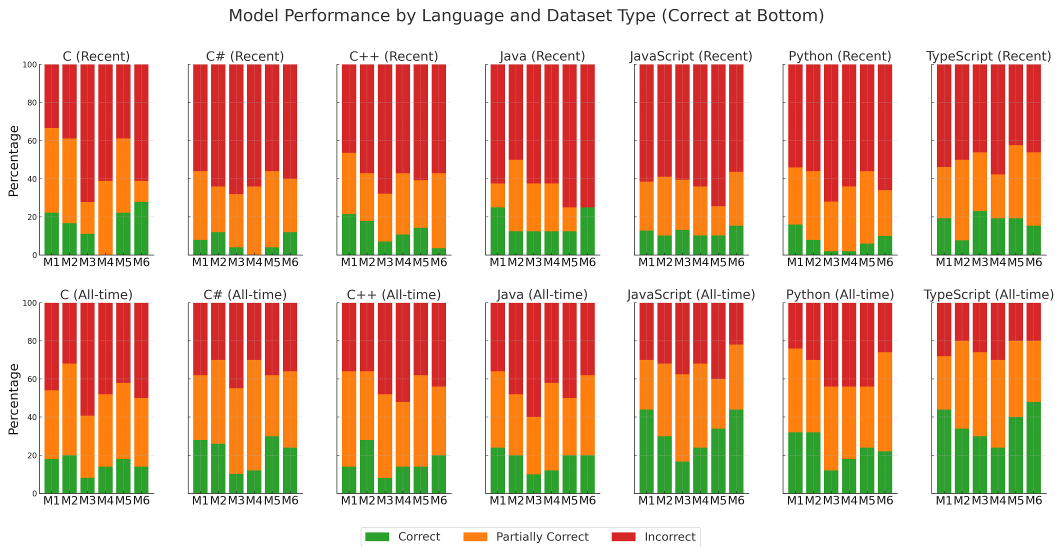

Figure 6: Cumulative comparison of model performance across seven programming languages. Each bar represents a model's prediction outcome distribution on GitHub issues, broken down into **Correct (green)**, **Partially Correct (orange)**, and **Incorrect (red)** responses. The top row corresponds to 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.

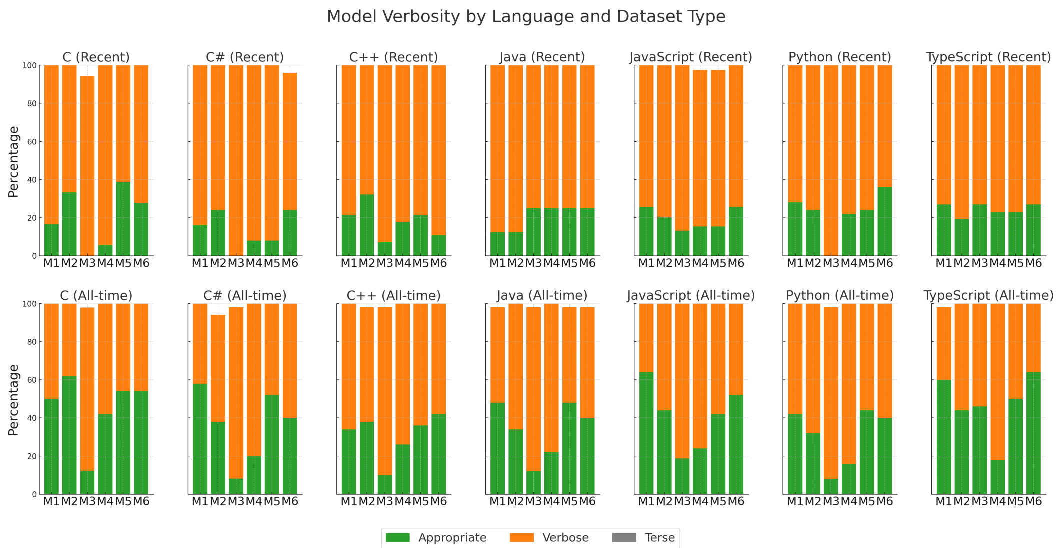

Figure 7: Cumulative verbosity distribution of model responses across seven programming languages. Each bar shows the proportion of responses classified as **Appropriate** (green), **Verbose** (orange), and **Terse** (gray, rarely observed). The top row corresponds to evaluations on **recent repositories**, 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 maintainer'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"

> **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 4.5. 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

Dataset Type	Correct	Partial	Incorrect	Total Accuracy
All-time repos	25.71%	35.43%	38.86%	61.14%
Recent repos	16.49%	30.41%	53.09%	46.90%
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.

J.4 Implications for Benchmark Design

These findings have important implications for understanding model capabilities:

1. **Training Data Recency Matters**: The 57.51 percentage point difference between recent repositories (16.49%) 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 310 AI responses used in 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 4.5 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 310 analyzed issues

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 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.