

Developer-LLM Conversations: An Empirical Study of Interactions and Generated Code Quality

Suzhen Zhong, Ying Zou, *Senior Member, IEEE*, Bram Adams, *Senior Member, IEEE*

Abstract—Large Language Models (LLMs) are becoming integral to modern software development workflows, assisting developers with code generation, API explanation, and iterative problem-solving through natural language conversations. Despite widespread adoption, there is limited empirical understanding of how developers interact with LLMs in practice and how these conversational dynamics influence task outcomes, code quality, and software engineering workflows. To address this, we systematically analyze developer-LLM conversation structures, developer behavior patterns, and LLM-generated code quality, uncovering key insights into LLM-assisted software development. For this analysis, we leverage CodeChat [83], a large dataset comprising 82,845 real-world developer-LLM conversations, containing 368,506 code snippets generated across over 20 programming languages, derived from the WildChat dataset [80]. We find that LLM responses are substantially longer than developer prompts, with a median token-length ratio of 14:1. Multi-turn conversations account for 68% of the dataset and often evolve due to shifting requirements, incomplete prompts, or clarification requests. Topic analysis identifies web design (9.6% of conversations) and neural network training (8.7% of conversations) as the most frequent LLM-assisted tasks. Evaluation across five languages (i.e., Python, JavaScript, C++, Java, and C#) reveals prevalent and language-specific issues in LLM-generated code: generated Python and JavaScript code often include undefined variables (83.4% and 75.3% of code snippets, respectively); Java code lacks required comments (75.9%); C++ code frequently omits headers (41.1%) and C# code shows unresolved namespaces (49.2%). During a conversation, syntax and import errors persist across turns; however, documentation quality in Java improves by up to 14.7%, and import handling in Python improves by 3.7% over 5 turns. Prompts that point out mistakes in code generated in prior turns and explicitly request a fix are most effective for resolving errors.

Index Terms—Developer-LLMs Conversations, Code Generation, CodeChat Dataset, Code Quality

I. INTRODUCTION

Large Language Models (LLMs), such as ChatGPT [46] and Claude [4], trained on large corpora of text and code [73], aim to understand developer intent and provide functionally relevant responses. Developers increasingly engage with LLMs through conversational prompts to perform tasks such as code generation [33], [27] and code quality evaluation [66], [28]. These interactions typically unfold as *conversations*, composed of a sequence of prompt–response pairs (i.e., turns) between a developer and the LLM. Conversations may be *single-turn*,

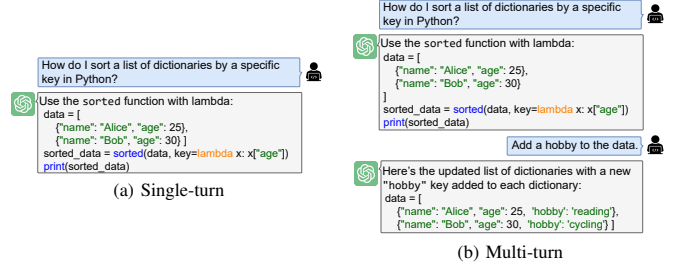


Fig. 1: Examples of developer-LLM conversations.

addressing isolated tasks, or *multi-turn*, involving iterative refinement based on prior context, as illustrated in Figure 1, which contrasts a single-turn exchange with a multi-turn, iterative dialogue.

However, detailed conversational data from commercial LLM services such as OpenAI and Anthropic remains unavailable to the public due to privacy concerns [48] and intellectual property protections [5]. This limitation restricts comprehensive analyses and empirical studies. To address the lack of publicly accessible, prior research [76] has adopted methods to mine explicitly shared ChatGPT conversations from GitHub and Hacker News. While these methods provide valuable insights into LLM-driven code generation and workflows, they primarily rely on version control and issue-tracking artifacts (e.g., commits, issues, or pull requests), which tend to highlight finalized solutions or structured development tasks. In contrast, daily programming interactions typically involve iterative, exploratory, and frequently undocumented exchanges with LLMs, such as trial-and-error cycles, informal queries, and intermediate debugging conversations.

This analysis is performed on **CodeChat**, a large-scale dataset of real-world developer-LLM conversations that we obtained by filtering the WildChat dataset [80], which comprises interactions mined from publicly accessible chatbot services. CodeChat comprises 82,845 developer-LLM conversations, totalling 311,161 turns and 368,506 code snippets across more than 20 programming languages.

This paper focuses on understanding the conversational characteristics, topical trends, and the quality of LLM-generated code of code-centric developer-LLM interactions. Specifically, we investigate the following research questions (RQs):

RQ1. What are the characteristics of developer-LLM conversations in CodeChat?

Understanding the characteristics of developer-LLM conversations provides foundational insights into how

Suzhen Zhong and Ying Zou are with the Department of Electrical and Computer Engineering, Queen's University, Kingston, ON K7L 3N6, Canada. E-mail: {suzhen.zhong, ying.zou}@queensu.ca.

Bram Adams is with the Maintenance, Construction and Intelligence of Software Lab (MCIS), School of Computing, Queen's University, Kingston, ON K7L 3N6, Canada. E-mail: bram.adams@queensu.ca.

developers engage with and receive responses from LLMs. In CodeChat, LLM responses are significantly longer than developer prompts, with a median token length ratio of 14:1. Compared to developer-generated answers on Stack Overflow, LLM responses are also more verbose, averaging approximately 2,000 characters per response, while Stack Overflow answers average 836 characters [43]. In terms of conversational dynamics, 68% of conversations are multi-turn, often because developers introduce new use cases, clarify missing details, or request additional features as the interaction progresses. LLMs generate code in over 20 languages, with 16% of LLM responses containing multi-language code, with common pairs like CSS-HTML and JavaScript-HTML reflecting real-world usage.

RQ2. What topics do developers most commonly prompt when interacting with LLMs?

Analyzing the common topics from developer-LLM interactions reveal the needs and challenges developers face, and guide the development of LLMs to better support real-world coding tasks. We apply BERTopic [24], a transformer-based topic modeling technique, to 52,086 English prompts from CodeChat, identifying 47 distinct topics. The most prevalent topics are web design (9.6% of conversations) and machine learning (8.7%), with Python being used in 72.9% of LLM-generated code for ML tasks, and HTML (39.0%) and JavaScript (26.6%) frequently used in web-related conversations. Developer-LLM engagement varies by topic, with AI-augmented business tool automation showing the highest interaction (mean = 3.40 turns). Repeated shifts to different use cases, alternation between adding features and changing goals, and consecutive LLM-generated errors without effective correction contribute to the longer interactions.

RQ3. How high is the quality of LLM-generated code within coding conversations?

Evaluating the quality of LLM-generated code highlights its strengths and shortcomings, offering insights to improve the documentation, performance, syntax correctness, and adherence to best practices of LLM-generated code. Across five major languages (i.e., Python, JavaScript, C++, Java, C#), we find widespread defects in the first-turn LLM-generated code. Undefined variables (e.g., 75.3% of code snippets in JavaScript, 30.8% in Python) and missing documentation (e.g., 75.9% of code snippets in Java) are particularly prevalent, while syntax errors are comparatively less frequent (e.g., 14.4% of code snippets in JavaScript, 8.0% in Python). These issues limit the effectiveness of LLM-generated code for direct use in real-world programming scenarios. In multi-turn conversations with Python, undefined variable errors increase from 23.5% of conversations to 32.8% after 5 turns, while import-related errors decrease from 48.3% to 44.6%, both changes being statistically significant ($p < 0.05$, with a low effect size). In Java, documentation violations

decline significantly from 78.1% to 63.4% of conversations ($p < 0.05$). The LLM user gets these syntax errors resolved most commonly by pointing out the mistake and requesting a fix (22.8%), by asking targeted questions (16.9%), or by adding specific instructions (16.5%). These findings indicate that clear error signaling, guided questioning, and precise directives are prevalent in successful corrections.

The main contributions of this paper are as follows:

1. We provide CodeChat, a comprehensive dataset of real-world developer-LLM conversations obtained from WildChat [80].
2. We characterize developer-LLM interaction dynamics by defining and applying conversation-level metrics that reveal key patterns in verbosity, turn structure, and prompt design. The metrics form the basis for identifying inefficiencies and guiding improvements in prompt engineering and interaction design.
3. We categorize the topics of developer-LLM interactions and analyze how the number of conversational turns and language distribution differ across tasks, providing actionable insights for developing more context-aware, task-specific LLM-assisted software engineering tools.
4. We perform a large-scale, multi-language analysis of LLM-generated code quality, uncovering widespread defects, language-specific error patterns, and limitations that affect correctness, maintainability, and usability. Our findings underscore the need for systematic quality control and enhanced tool support in LLM-assisted software development workflows.

To support reproducibility, we release the CodeChat dataset [83] on Hugging Face and make our code and analysis scripts available in a public replication package [67].

II. RELATED WORK

In this section, we present the work most closely related to developer-LLMs conversations on code-related tasks.

A. Datasets for Developer-LLMs Conversations

Large Language Models (LLMs) have significantly enhanced programming-related tasks. Due to the limited availability of conversational datasets capturing realistic developer interactions with LLMs, prior studies have relied on datasets collected from structured competitive programming sites, Q&A forums, social media platforms, and collaborative developer platforms. For example, Coignon et al. [14] and Nikolaidis et al. [44] evaluate LLMs using competitive programming problems from LeetCode, selecting questions that span different difficulty levels and programming languages. Additionally, Kabir et al. [34], Delile et al. [18], and Silva et al. [65] mine programming questions and solutions from Stack Overflow to assess performance of LLM-generated code. Furthermore, Feng et al. [21] analyze code snippets produced by LLMs and shared on social media platforms such as Twitter [8] and Reddit [56]. Xiao et al. [76] introduce a conversational dataset collected from GitHub [23] and Hacker

TABLE I: Overview of conversational datasets.

	#Convo	#Snippets	Platform	Time (Duration)
DevGPT	29,778	19,106	GitHub, Hacker News	2023.07 – 2023.10 (4 months)
GPT-Social	332	-	Tweet, Reddit	2023.12 – 2024.01 (2 months)
CodeChat	82,845	368,506	-	2023.04 – 2024.05 (14 months)

* The term “Convo” refers to “conversations”. Number of users: CodeChat 26,085, estimated from unique IP addresses; DevGPT and GPT-Social user counts not available.

News [77], comprising 29,778 prompts and 19,106 corresponding code snippets.

Unlike prior datasets, our work introduces CodeChat (as shown in Table I), which comprises 82,845 real-world developer-LLM conversations. In contrast to previous datasets, which are limited to curated platforms or self-reported content, CodeChat collects unconstrained and unedited dialogues from publicly accessible chatbot services, building upon the WildChat dataset [80]. By leveraging this naturalistic source, CodeChat provides a dataset of authentic and spontaneous interactions that reflect developers’ daily practical usage.

B. Analysis of Developers’ Prompts to LLMs

Previous studies have analyzed developers’ prompts based on data obtained from GitHub repositories. For example, Hao et al. [30] categorize 580 ChatGPT conversations extracted from GitHub pull requests and issues into 16 prompt types. Sagdic et al. [60] identify 17 distinct topics by examining 1,701 prompts collected from diverse GitHub activities. Furthermore, Tufano et al. [69] create a high-level taxonomy based on the analysis of 467 ChatGPT mentions within GitHub commits, pull requests, and issues. Additionally, Das et al. [17] investigate 1,152 conversations from 1,012 GitHub issues to explore how developers use ChatGPT.

In contrast, our work broadens the scope by leveraging the CodeChat dataset, which contains authentic conversational interactions between developers and LLMs. We analyze 52,086 developer prompts and reveal 47 distinct prompt topics. These common topics show that conversations shared on GitHub are often specialized and platform-centric, whereas developers’ prompts in CodeChat encompass a wider range of practical software engineering tasks.

C. Code Quality Issues of LLM-Generated Code

As LLM-generated code becomes more prevalent in software engineering, ensuring its quality is essential, since poor code can negatively impact reliability, performance, and maintainability. To recognize this concern, recent studies have empirically investigated the quality of LLM-generated code. For instance, Liu et al. [37] examine code quality issues in ChatGPT-generated solutions for 2,033 LeetCode coding tasks. Similarly, Siddiq et al. [64] use static analysis to identify code smells and security issues in Python and Java snippets generated by ChatGPT from the DevGPT dataset. Additionally,

Listing 1: Example of a conversation from the WildChat dataset

```

1 {
2   "conversation_id": "c9ec5b440fbdd2a264",
3   "model": "gpt-4-0314",
4   "conversation": [
5     {
6       "role": "user",
7       "content": "flutter 3.7 how to set background
8         color for disabled inputformtext() widget?"
9     },
10    {
11      "role": "LLM",
12      "content": "To set the background colour for a
13        disabled `InputFormText` widget in Flutter 3
14        .7, use the `decoration` property of `
15        InputDecoration` and set the `fillColor`.
16        \nHere's an example:
17        \n\n```\n
18        dart\n
19        nInputFormText(\n
20        enabled: false,\n
21        decoration: InputDecoration(\n
22        filled: true\n
23        ,\n
24        fillColor: Colors.grey[200], // Set
25        background colour\n
26        \n labelText: 'Disabled
27        input',\n
28        ),\n
29        ),\n
30        ),\n
31        ),\n
32        ),\n
33        ),\n
34        ),\n
35        ),\n
36        ),\n
37        ),\n
38        ),\n
39        ),\n
40        ),\n
41        ),\n
42        ),\n
43        ),\n
44        ),\n
45        ),\n
46        ),\n
47        ),\n
48        ),\n
49        ),\n
50        ),\n
51        ),\n
52        ),\n
53        ),\n
54        ),\n
55        ),\n
56        ),\n
57        ),\n
58        ),\n
59        ),\n
60        ),\n
61        ),\n
62        ),\n
63        ),\n
64        ),\n
65        ),\n
66        ),\n
67        ),\n
68        ),\n
69        ),\n
70        ),\n
71        ),\n
72        ),\n
73        ),\n
74        ),\n
75        ),\n
76        ),\n
77        ),\n
78        ),\n
79        ),\n
80        ),\n
81        ),\n
82        ),\n
83        ),\n
84        ),\n
85        ),\n
86        ),\n
87        ),\n
88        ),\n
89        ),\n
90        ),\n
91        ),\n
92        ),\n
93        ),\n
94        ),\n
95        ),\n
96        ),\n
97        ),\n
98        ),\n
99        ),\n
100       ),\n
101       ),\n
102       ),\n
103       ),\n
104       ),\n
105       ),\n
106       ),\n
107       ),\n
108       ),\n
109       ),\n
110       ),\n
111       ),\n
112       ),\n
113       ),\n
114       ),\n
115       ),\n
116       ),\n
117       ),\n
118       ),\n
119       ),\n
120       ),\n
121       ),\n
122       ),\n
123       ),\n
124       ),\n
125       ),\n
126       ),\n
127       ),\n
128       ),\n
129       ),\n
130       ),\n
131       ),\n
132       ),\n
133       ),\n
134       ),\n
135       ),\n
136       ),\n
137       ),\n
138       ),\n
139       ),\n
140       ),\n
141       ),\n
142       ),\n
143       ),\n
144       ),\n
145       ),\n
146       ),\n
147       ),\n
148       ),\n
149       ),\n
150       ),\n
151       ),\n
152       ),\n
153       ),\n
154       ),\n
155       ),\n
156       ),\n
157       ),\n
158       ),\n
159       ),\n
160       ),\n
161       ),\n
162       ),\n
163       ),\n
164       ),\n
165       ),\n
166       ),\n
167       ),\n
168       ),\n
169       ),\n
170       ),\n
171       ),\n
172       ),\n
173       ),\n
174       ),\n
175       ),\n
176       ),\n
177       ),\n
178       ),\n
179       ),\n
180       ),\n
181       ),\n
182       ),\n
183       ),\n
184       ),\n
185       ),\n
186       ),\n
187       ),\n
188       ),\n
189       ),\n
190       ),\n
191       ),\n
192       ),\n
193       ),\n
194       ),\n
195       ),\n
196       ),\n
197       ),\n
198       ),\n
199       ),\n
200       ),\n
201       ),\n
202       ),\n
203       ),\n
204       ),\n
205       ),\n
206       ),\n
207       ),\n
208       ),\n
209       ),\n
210       ),\n
211       ),\n
212       ),\n
213       ),\n
214       ),\n
215       ),\n
216       ),\n
217       ),\n
218       ),\n
219       ),\n
220       ),\n
221       ),\n
222       ),\n
223       ),\n
224       ),\n
225       ),\n
226       ),\n
227       ),\n
228       ),\n
229       ),\n
230       ),\n
231       ),\n
232       ),\n
233       ),\n
234       ),\n
235       ),\n
236       ),\n
237       ),\n
238       ),\n
239       ),\n
240       ),\n
241       ),\n
242       ),\n
243       ),\n
244       ),\n
245       ),\n
246       ),\n
247       ),\n
248       ),\n
249       ),\n
250       ),\n
251       ),\n
252       ),\n
253       ),\n
254       ),\n
255       ),\n
256       ),\n
257       ),\n
258       ),\n
259       ),\n
260       ),\n
261       ),\n
262       ),\n
263       ),\n
264       ),\n
265       ),\n
266       ),\n
267       ),\n
268       ),\n
269       ),\n
270       ),\n
271       ),\n
272       ),\n
273       ),\n
274       ),\n
275       ),\n
276       ),\n
277       ),\n
278       ),\n
279       ),\n
280       ),\n
281       ),\n
282       ),\n
283       ),\n
284       ),\n
285       ),\n
286       ),\n
287       ),\n
288       ),\n
289       ),\n
290       ),\n
291       ),\n
292       ),\n
293       ),\n
294       ),\n
295       ),\n
296       ),\n
297       ),\n
298       ),\n
299       ),\n
300       ),\n
301       ),\n
302       ),\n
303       ),\n
304       ),\n
305       ),\n
306       ),\n
307       ),\n
308       ),\n
309       ),\n
310       ),\n
311       ),\n
312       ),\n
313       ),\n
314       ),\n
315       ),\n
316       ),\n
317       ),\n
318       ),\n
319       ),\n
320       ),\n
321       ),\n
322       ),\n
323       ),\n
324       ),\n
325       ),\n
326       ),\n
327       ),\n
328       ),\n
329       ),\n
330       ),\n
331       ),\n
332       ),\n
333       ),\n
334       ),\n
335       ),\n
336       ),\n
337       ),\n
338       ),\n
339       ),\n
340       ),\n
341       ),\n
342       ),\n
343       ),\n
344       ),\n
345       ),\n
346       ),\n
347       ),\n
348       ),\n
349       ),\n
350       ),\n
351       ),\n
352       ),\n
353       ),\n
354       ),\n
355       ),\n
356       ),\n
357       ),\n
358       ),\n
359       ),\n
360       ),\n
361       ),\n
362       ),\n
363       ),\n
364       ),\n
365       ),\n
366       ),\n
367       ),\n
368       ),\n
369       ),\n
370       ),\n
371       ),\n
372       ),\n
373       ),\n
374       ),\n
375       ),\n
376       ),\n
377       ),\n
378       ),\n
379       ),\n
380       ),\n
381       ),\n
382       ),\n
383       ),\n
384       ),\n
385       ),\n
386       ),\n
387       ),\n
388       ),\n
389       ),\n
390       ),\n
391       ),\n
392       ),\n
393       ),\n
394       ),\n
395       ),\n
396       ),\n
397       ),\n
398       ),\n
399       ),\n
400       ),\n
401       ),\n
402       ),\n
403       ),\n
404       ),\n
405       ),\n
406       ),\n
407       ),\n
408       ),\n
409       ),\n
410       ),\n
411       ),\n
412       ),\n
413       ),\n
414       ),\n
415       ),\n
416       ),\n
417       ),\n
418       ),\n
419       ),\n
420       ),\n
421       ),\n
422       ),\n
423       ),\n
424       ),\n
425       ),\n
426       ),\n
427       ),\n
428       ),\n
429       ),\n
430       ),\n
431       ),\n
432       ),\n
433       ),\n
434       ),\n
435       ),\n
436       ),\n
437       ),\n
438       ),\n
439       ),\n
440       ),\n
441       ),\n
442       ),\n
443       ),\n
444       ),\n
445       ),\n
446       ),\n
447       ),\n
448       ),\n
449       ),\n
450       ),\n
451       ),\n
452       ),\n
453       ),\n
454       ),\n
455       ),\n
456       ),\n
457       ),\n
458       ),\n
459       ),\n
460       ),\n
461       ),\n
462       ),\n
463       ),\n
464       ),\n
465       ),\n
466       ),\n
467       ),\n
468       ),\n
469       ),\n
470       ),\n
471       ),\n
472       ),\n
473       ),\n
474       ),\n
475       ),\n
476       ),\n
477       ),\n
478       ),\n
479       ),\n
480       ),\n
481       ),\n
482       ),\n
483       ),\n
484       ),\n
485       ),\n
486       ),\n
487       ),\n
488       ),\n
489       ),\n
490       ),\n
491       ),\n
492       ),\n
493       ),\n
494       ),\n
495       ),\n
496       ),\n
497       ),\n
498       ),\n
499       ),\n
500       ),\n
501       ),\n
502       ),\n
503       ),\n
504       ),\n
505       ),\n
506       ),\n
507       ),\n
508       ),\n
509       ),\n
510       ),\n
511       ),\n
512       ),\n
513       ),\n
514       ),\n
515       ),\n
516       ),\n
517       ),\n
518       ),\n
519       ),\n
520       ),\n
521       ),\n
522       ),\n
523       ),\n
524       ),\n
525       ),\n
526       ),\n
527       ),\n
528       ),\n
529       ),\n
530       ),\n
531       ),\n
532       ),\n
533       ),\n
534       ),\n
535       ),\n
536       ),\n
537       ),\n
538       ),\n
539       ),\n
540       ),\n
541       ),\n
542       ),\n
543       ),\n
544       ),\n
545       ),\n
546       ),\n
547       ),\n
548       ),\n
549       ),\n
550       ),\n
551       ),\n
552       ),\n
553       ),\n
554       ),\n
555       ),\n
556       ),\n
557       ),\n
558       ),\n
559       ),\n
560       ),\n
561       ),\n
562       ),\n
563       ),\n
564       ),\n
565       ),\n
566       ),\n
567       ),\n
568       ),\n
569       ),\n
570       ),\n
571       ),\n
572       ),\n
573       ),\n
574       ),\n
575       ),\n
576       ),\n
577       ),\n
578       ),\n
579       ),\n
580       ),\n
581       ),\n
582       ),\n
583       ),\n
584       ),\n
585       ),\n
586       ),\n
587       ),\n
588       ),\n
589       ),\n
590       ),\n
591       ),\n
592       ),\n
593       ),\n
594       ),\n
595       ),\n
596       ),\n
597       ),\n
598       ),\n
599       ),\n
600       ),\n
601       ),\n
602       ),\n
603       ),\n
604       ),\n
605       ),\n
606       ),\n
607       ),\n
608       ),\n
609       ),\n
610       ),\n
611       ),\n
612       ),\n
613       ),\n
614       ),\n
615       ),\n
616       ),\n
617       ),\n
618       ),\n
619       ),\n
620       ),\n
621       ),\n
622       ),\n
623       ),\n
624       ),\n
625       ),\n
626       ),\n
627       ),\n
628       ),\n
629       ),\n
630       ),\n
631       ),\n
632       ),\n
633       ),\n
634       ),\n
635       ),\n
636       ),\n
637       ),\n
638       ),\n
639       ),\n
640       ),\n
641       ),\n
642       ),\n
643       ),\n
644       ),\n
645       ),\n
646       ),\n
647       ),\n
648       ),\n
649       ),\n
650       ),\n
651       ),\n
652       ),\n
653       ),\n
654       ),\n
655       ),\n
656       ),\n
657       ),\n
658       ),\n
659       ),\n
660       ),\n
661       ),\n
662       ),\n
663       ),\n
664       ),\n
665       ),\n
666       ),\n
667       ),\n
668       ),\n
669       ),\n
670       ),\n
671       ),\n
672       ),\n
673       ),\n
674       ),\n
675       ),\n
676       ),\n
677       ),\n
678       ),\n
679       ),\n
680       ),\n
681       ),\n
682       ),\n
683       ),\n
684       ),\n
685       ),\n
686       ),\n
687       ),\n
688       ),\n
689       ),\n
690       ),\n
691       ),\n
692       ),\n
693       ),\n
694       ),\n
695       ),\n
696       ),\n
697       ),\n
698       ),\n
699       ),\n
700       ),\n
701       ),\n
702       ),\n
703       ),\n
704       ),\n
705       ),\n
706       ),\n
707       ),\n
708       ),\n
709       ),\n
710       ),\n
711       ),\n
712       ),\n
713       ),\n
714       ),\n
715       ),\n
716       ),\n
717       ),\n
718       ),\n
719       ),\n
720       ),\n
721       ),\n
722       ),\n
723       ),\n
724       ),\n
725       ),\n
726       ),\n
727       ),\n
728       ),\n
729       ),\n
730       ),\n
731       ),\n
732       ),\n
733       ),\n
734       ),\n
735       ),\n
736       ),\n
737       ),\n
738       ),\n
739       ),\n
740       ),\n
741       ),\n
742       ),\n
743       ),\n
744       ),\n
745       ),\n
746       ),\n
747       ),\n
748       ),\n
749       ),\n
750       ),\n
751       ),\n
752       ),\n
753       ),\n
754       ),\n
755       ),\n
756       ),\n
757       ),\n
758       ),\n
759       ),\n
760       ),\n
761       ),\n
762       ),\n
763       ),\n
764       ),\n
765       ),\n
766       ),\n
767       ),\n
768       ),\n
769       ),\n
770       ),\n
771       ),\n
772       ),\n
773       ),\n
774       ),\n
775       ),\n
776       ),\n
777       ),\n
778       ),\n
779       ),\n
780       ),\n
781       ),\n
782       ),\n
783       ),\n
784       ),\n
785       ),\n
786       ),\n
787       ),\n
788       ),\n
789       ),\n
790       ),\n
791       ),\n
792       ),\n
793       ),\n
794       ),\n
795       ),\n
796       ),\n
797       ),\n
798       ),\n
799       ),\n
800       ),\n
801       ),\n
802       ),\n
803       ),\n
804       ),\n
805       ),\n
806       ),\n
807       ),\n
808       ),\n
809       ),\n
810       ),\n
811       ),\n
812       ),\n
813       ),\n
814       ),\n
815       ),\n
816       ),\n
817       ),\n
818       ),\n
819       ),\n
820       ),\n
821       ),\n
822       ),\n
823       ),\n
824       ),\n
825       ),\n
826       ),\n
827       ),\n
828       ),\n
829       ),\n
830       ),\n
831       ),\n
832       ),\n
833       ),\n
834       ),\n
835       ),\n
836       ),\n
837       ),\n
838       ),\n
839       ),\n
840       ),\n
841       ),\n
842       ),\n
843       ),\n
844       ),\n
845       ),\n
846       ),\n
847       ),\n
848       ),\n
849       ),\n
850       ),\n
851       ),\n
852       ),\n
853       ),\n
854       ),\n
855       ),\n
856       ),\n
857       ),\n
858       ),\n
859       ),\n
860       ),\n
861       ),\n
862       ),\n
863       ),\n
864       ),\n
865       ),\n
866       ),\n
867       ),\n
868       ),\n
869       ),\n
870       ),\n
871       ),\n
872       ),\n
873       ),\n
874       ),\n
875       ),\n
876       ),\n
877       ),\n
878       ),\n
879       ),\n
880       ),\n
881       ),\n
882       ),\n
883       ),\n
884       ),\n
885       ),\n
886       ),\n
887       ),\n
888       ),\n
889       ),\n
890       ),\n
891       ),\n
892       ),\n
893       ),\n
894       ),\n
895       ),\n
896       ),\n
897       ),\n
898       ),\n
899       ),\n
900       ),\n
901       ),\n
902       ),\n
903       ),\n
904       ),\n
905       ),\n
906       ),\n
907       ),\n
908       ),\n
909       ),\n
910       ),\n
911       ),\n
912       ),\n
913       ),\n
914       ),\n
915       ),\n
916       ),\n
917       ),\n
918       ),\n
919       ),\n
920       ),\n
921       ),\n
922       ),\n
923       ),\n
924       ),\n
925       ),\n
926       ),\n
927       ),\n
928       ),\n
929       ),\n
930       ),\n
931       ),\n
932       ),\n
933       ),\n
934       ),\n
935       ),\n
936       ),\n
937       ),\n
938       ),\n
939       ),\n
940       ),\n
941       ),\n
942       ),\n
943       ),\n
944       ),\n
945       ),\n
946       ),\n
947       ),\n
948       ),\n
949       ),\n
950       ),\n
951       ),\n
952       ),\n
953       ),\n
954       ),\n
955       ),\n
956       ),\n
957       ),\n
958       ),\n
959       ),\n
960       ),\n
961       ),\n
962       ),\n
963       ),\n
964       ),\n
965       ),\n
966       ),\n
967       ),\n
968       ),\n
969       ),\n
970       ),\n
971       ),\n
972       ),\n
973       ),\n
974       ),\n
975       ),\n
976       ),\n
977       ),\n
978       ),\n
979       ),\n
980       ),\n
981       ),\n
982       ),\n
983       ),\n
984       ),\n
985       ),\n
986       ),\n
987       ),\n
988       ),\n
989       ),\n
990       ),\n
991       ),\n
992       ),\n
993       ),\n
994       ),\n
995       ),\n
996       ),\n
997       ),\n
998       ),\n
999       ),\n
1000      ),\n

```

Almanasra and Suwais [1] evaluate documentation, exception handling, and unused variables in code generated by ChatGPT-4o across 500 programming tasks.

In contrast to these studies, we conduct a larger-scale analysis of 63,685 code snippets across five programming languages (i.e., Python, JavaScript, C++, Java, and C#), drawn from real-world developer-LLM conversations. Beyond single-turn conversations, we analyze how code quality issues evolve over consecutive turns, allowing us to track the emergence and persistence of common issues as developers iteratively interact with LLMs in practical scenarios.

III. METHODOLOGY AND DATA COLLECTION

In this section, we provide an overview of our methodology for analyzing LLM usage in software development tasks. As shown in Figure 2, we first extract the software engineering specific CodeChat dataset from the existing WildChat dataset [80]. Then, our analysis proceeds in three stages. First, we examine the characteristics of the CodeChat dataset by computing conversation-level metrics. Next, we select English prompts and apply topic modeling to uncover common developer intents. Finally, we focus on the five most frequently used programming languages and assess the quality of LLM-generated code within multi-turn conversations. While the methodological details of the analyses are discussed in the next section, here we discuss in detail how CodeChat was derived.

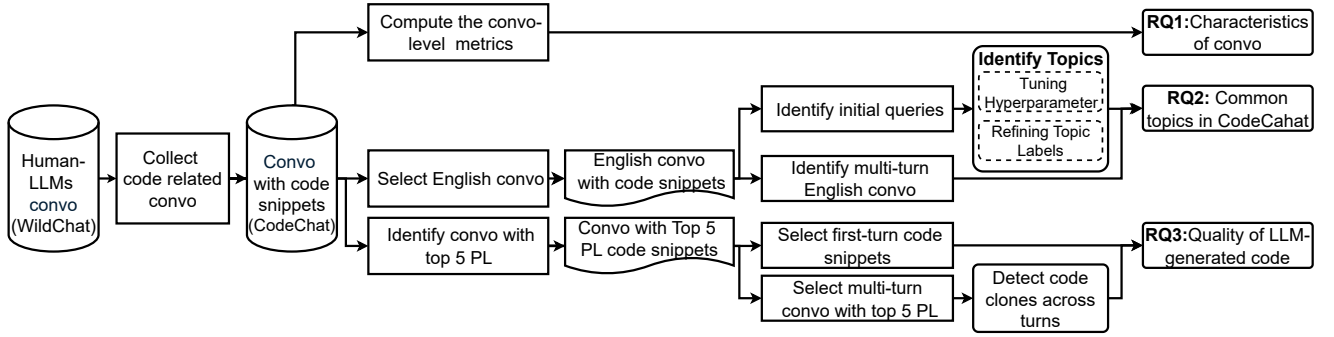


Fig. 2: Overview of the approach. The term “convo” refers to “conversations”, and “PL” in the figure stands for “Programming Language”.

A. Data Processing

The dataset that we use is derived from the recent WildChat [80], a large-scale collection of 837,989 real-world human-LLM conversations comprising 1,960,074 conversational turns. Collected from April 2023 to May 2024 across 68 languages and 204,736 users, it reflects recent and diverse ChatGPT usage via a public interface on Hugging Face [75]. Compared to LMSYS-Chat-1M [81], which includes about 70% Vicuna [10] interactions, WildChat is more relevant due to its focus on ChatGPT, a widely adopted model. OpenAssistant [35], in contrast, ends data collection in April 2023 and contains 66,479 conversations, making WildChat (with 837,989 conversations) both more current and substantially larger.

As shown in Listing 1, conversations in WildChat are represented in a JSON format defined by the dataset creators to support systematic analysis and reproducibility. A WildChat entry contains metadata such as `conversation_id` and `model`, along with a `conversation` field containing a list of dialogue turns. Each turn includes attributes such as `role` (e.g., user or LLM), `content`, `language`, `timestamp`, and `hashed_ip`.

From WildChat, we extract the subset of all conversations that include code snippets, as these are directly relevant to software development tasks. We identify code-related conversations by examining the `content` field of each LLM turn. If that field contains code enclosed in triple backticks (“`````”), a standard Markdown format for code blocks, the conversation is retained. This filtering process yields 82,845 conversations (9.9% of WildChat), comprising 311,161 conversational turns and 368,506 code snippets. The resulting dataset, which we refer to as CodeChat, captures a diverse range of real-world coding interactions contributed by 26,085 unique users, as determined by distinct `hashed_ip` values. In CodeChat, we treat user in a code related conversations as `developer` to clarify participant roles. All other metadata from WildChat is inherited without changes, preserving context and reproducibility for further analysis.

IV. RESULTS

For each RQ, the section presents its motivation, approach, and findings.

TABLE II: Definition of conversation-level metrics

Metric (Abbreviation)	Definition
Token Ratio (TR)	Ratio of LLM response token count to developer prompt token count in a conversation.
Turn Count (TC)	Total number of prompt-response pairs (turns) in a conversation.
Single-turn Conversation Count (STC-Count)	Number of conversations consisting of only one turn.
Multi-turn Conversation Count (MTC-Count)	Number of conversations consisting of more than one turn.
Prompt Design Gap Frequency (PDG-Freq{ X })	Number of occurrences of prompt design gap type X between consecutive developer prompts in multi-turn conversations.
Programming Language Rate (PL-Rate{ X })	Proportion of LLM responses generating code in language X .
Lines of Code (LOC)	Number of non-blank lines in each LLM-generated code snippet.
Multi-language Co-occur Rate (MLC-Rate{ X - Y })	Percentage of conversations that include code in both programming languages X and Y , calculated as $\frac{\text{count}(X \cap Y)}{\text{count}(X) + \text{count}(Y)}$

A. RQ1: What are the characteristics of developer-LLM conversations in CodeChat?

1) *Motivation:* Developers increasingly rely on conversations with LLMs to solve problems, refine their understanding, and improve overall productivity. Despite the growing use of LLMs in real-world coding scenarios, little is known about the characteristics of these interactions. In this RQ, we aim to uncover how developers engage with LLMs and how LLMs respond. To provide foundational insights for developers, conversational code assistants, researchers, and tool builders who integrate conversational interfaces into IDEs. Our findings will allow them to better understand and improve LLMs, ultimately supporting more effective and efficient programming workflows.

2) *Approach:* (a) *Definition of Conversation-level Metrics.* To quantitatively characterize developer-LLM interactions, we define conversation-level metrics that capture different aspects of conversational efficiency, structure, and generated code properties, as listed in Table II. The metrics are designed to measure the following five aspects:

Measuring token utilization efficiency. Efficient token usage is critical in developer-LLM conversations, as output

tokens (e.g., GPT-4o) cost significantly more than inputs, and excessive verbosity reduces interaction efficiency. We define the *Token Ratio (TR)* as the number of LLM-generated tokens divided by the number of developer prompt tokens, using the GPT-4 tokenizer [49] for consistency. To compare token lengths between developer prompts and LLM responses across CodeChat, we employ the Wilcoxon signed-rank test [59], a non-parametric statistical test for paired samples, with a significance level of $p = 0.05$. Additionally, we compare the average number of characters in LLM-generated responses with those in human-written answers on Stack Overflow to assess how LLM verbosity aligns with typical developer communication.

Analyzing conversational structure and prompt design gaps. To characterize structural dynamics, we use *Turn Count (TC)* to measure the number of prompt–response pairs in each conversation. Based on *TC*, each conversation is classified as single-turn or multi-turn. *STC-Count* and *MTC-Count* denote the total counts in each category. Multi-turn conversations often result from prompt deficiencies that require clarification or revision, increasing *TC* and token cost.

To investigate the causes of increased *Turn Count* in multi-turn conversations, we extract pairs of consecutive developer prompts within multi-turn conversations. Each prompt pair serves to identify potential deficiencies in the initial prompt that lead to follow-up clarification or revision. The *Prompt Design Gap Frequency (PDG-Freq_X)* is defined as the proportion of prompt pairs of type *X*, where gap types are classified according to the framework proposed by Mondal et al. [40]. This metric quantifies inefficient multi-turn interactions and offers actionable guidance for prompt engineering.

Assessing programming language generation patterns. We use *Programming Language Rate (PL-Rate_X)* to quantify the proportion of LLM-generated code snippets tagged with each language *X* in CodeChat dataset. From the 368,506 total snippets, we retain 278,251 (75.5%) with explicit language tags and normalize common aliases (e.g., `js` to `javascript`, `c++` to `cpp`). We compute *PL-Rate_X* for each programming language *X* and identify the 20 most frequently generated languages. To assess alignment with human programming behaviour, we compare these distributions to the TIOBE Programming Community Index (January 2025) [49]. This comparison reveals potential generation biases in LLM outputs and offers insight into coverage across real-world programming languages.

Evaluating code snippet size using lines of code (LOC). To examine the size and complexity of LLM-generated code, we measure the *Lines of Code (LOC)* of each generated code snippet, by counting the number of non-blank lines. The LOC metric allows systematic comparison of code snippet sizes across programming languages and supports further analysis of language-specific generation behaviors.

Detecting multilingual code generation. To examine cross-language patterns in LLM-generated code, we compute the *Multi-language Co-occurrence Rate (MLC-Rate_{X-Y})*, defined as the percentage of conversations that include both *X* and *Y*. If a conversation contains Python, Bash, and Java, it contributes one count each to Python–Bash, Python–Java, and Bash–Java. This metric reflects the relative prevalence of

language integration in developer–LLM interactions.

(b) *Verification of Prompt Design Gap Annotations.* To systematically identify deficiencies in developer prompts that lead to multi-turn LLM conversations, we randomly sample 400 conversations from the 56,322 multi-turn interactions in CodeChat, based on standard sample size estimation (95% confidence level, 5% margin of error). For each sampled conversation, we randomly extract one pair of consecutive developer prompts (i.e., two adjacent turns), resulting in 400 prompt pairs for gap type annotation.

The annotation process is based on the prompt design gap framework proposed by Mondal et al. [40], which defines 11 gap types (as shown in Figure 4(b)) that commonly arise when user instructions are ambiguous, incomplete, or misaligned with intent. Annotation proceeds in two stages. In the first stage, each prompt pair is automatically labeled with one of the 11 gap types using ChatGPT-4o-mini, which is supplied with the full set of definitions and one-shot examples from the original framework to ensure consistency and accuracy. For example, adding a previously unstated requirement results in a “Missing Specification” label.

In the second stage, the first author independently annotated the same 400 prompt pairs to assess the reliability of the LLM-based labels. The resulting Cohen’s Kappa [74] is 0.81, indicating substantial agreement and supporting the use of LLM-based annotation for large-scale analysis.

(c) *Hypothesis Formulation and Statistical Analysis.* To assess whether LLM-generated code snippet sizes differ by programming language, we test the following hypotheses:

Null Hypothesis (H_0): There is no significant difference in the lines of code (LOC) of LLM-generated code snippets among the top 20 programming languages.

Alternative Hypothesis (H_A): There is a significant difference in the lines of code (LOC) among the top 20 programming languages.

We analyze code snippets corresponding to the top 20 programming languages by PL-Rate. For each language, we collect all lines of code (LOC) measurements from its code snippets, resulting in 20 language-specific LOC vectors. We then compare LOC distributions across these 20 vectors using the Kruskal–Wallis *H* test [72], a non-parametric method suitable for comparing multiple independent groups, with significance level $p = 0.05$. If a statistically significant overall difference is detected, we perform post-hoc pairwise comparisons using Dunn’s test with Bonferroni correction [19] to identify which language pairs differ significantly in snippet size. The statistical analysis provides statistical evidence to support or reject the hypothesis regarding language-specific variation in LLM-generated code snippet sizes.

3) *Findings: LLM responses in developer–LLM conversations are 2.4 times longer than human-written answers on Stack Overflow.* As shown in Figure 3, LLM-generated responses are significantly longer than the corresponding developer prompts in CodeChat conversations ($p < 0.05$), with a median Token Ratio (TR) of 14. On average, LLM responses

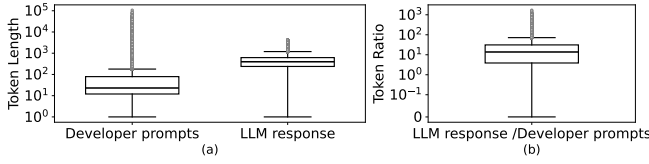


Fig. 3: (a) Token length distributions for developer prompts and LLM responses (log scale), and (b) Token ratio (TR) distribution with symmetric logarithmic scaling (symlog)

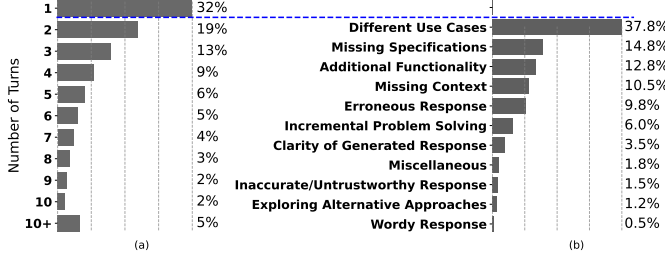


Fig. 4: (a) Distribution of conversation turns measured by Turn Count (TC) and (b) Frequency of Prompt Design Gap types (PDG-Freq_X) associated with multi-turn ($TC > 2$) developer-LLM conversations. The 11 gap types are adapted from the framework proposed by Mondal et al. [40].

in CodeChat contain 2,000 characters, compared to 971 for developer prompts; both are longer than the typical Stack Overflow answer (836 characters) or question (604 characters) [43]. While longer responses may improve completeness, they also elevate operational costs, as output tokens (e.g., GPT-4: \$20 USD per million output tokens) are more expensive than input tokens (5 \$USD per million input tokens). The findings indicate a potential trade-off between response length and operational cost, warranting further investigation.

In CodeChat, 68% of conversations comprise multiple turns, with the most common prompt design gap types (PDG-Freq) being “Different Use Cases” (37.8%), “Missing Specifications” (14.8%), and “Additional Functionality” (12.8%). As illustrated in Figure 4(a), TC values in multi-turn conversations span from 2 to more than 10, reflecting considerable variation in interaction length. As shown in Figure 4(b), we labeled prompt design gap types according to the framework of Mondal et al. [40], using ChatGPT-4o-mini for initial annotation followed by manual verification. The predominance of the “Different Use Cases” gap type reflects developers ask for LLMs to resolve different problems in a single conversation, instead of opening a new conversation. While “Missing Specifications” and “Additional Functionality” arise from incomplete initial prompts or requests for new features as the conversation progresses. These findings suggest that improved prompt design guidelines and prompt validation support could help developers formulate clearer queries, reduce unnecessary conversational turns.

LLM-generated code shows notable differences in language distribution for JavaScript, SQL, and Bash compared to real-world popularity rankings, but aligns more closely for Python and C++. As shown in Figure 5, we

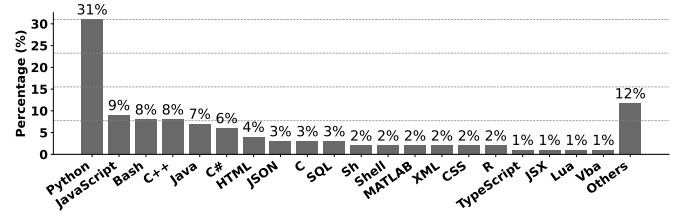


Fig. 5: Programming language rate (PL-Rate_X) for the top 20 languages generated by LLMs

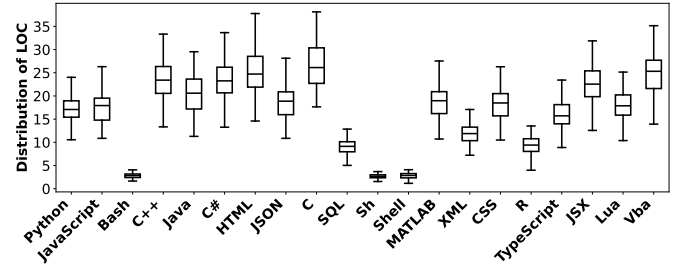


Fig. 6: Number of lines of code (LOC) distribution across the top 20 programming languages.

compute *PL-Rate* for the top 20 generated languages and compare them with their TIOBE rankings. JavaScript ranks second in CodeChat (9%, 23,534 out of 278,251), but only sixth in TIOBE. Similarly, TypeScript and JSX appear in the 17th and 18th positions, despite TypeScript ranking 37th and JSX being unlisted in TIOBE. SQL and Bash also show divergence: SQL ranks 10th in CodeChat (3%, 6,024 snippets) but 7th in TIOBE, while Bash is third in CodeChat but does not appear in TIOBE’s top 50. In contrast, Python and C++ exhibit more alignment, ranking first and fourth in CodeChat (31%, 84,924; and 8%, 20,021 snippets), and first and second in TIOBE, respectively. These disparities in programming language distribution align with prior work [70] showing that leading LLMs consistently favor Python and JavaScript, even when other languages are more appropriate.

LLM-generated code snippets are generally short, with a median LOC under 30, though substantial variation exists across programming languages. As shown in Figure 6, C (26 LOC), HTML (25 LOC), and C++ (23 LOC) produce the longest median snippet lengths. The Kruskal-Wallis H tests confirm significant differences between several of these languages, such as the generated code snippets in C being significantly longer than the ones in C++ ($p < 0.05$), and the snippets in C++ being significantly longer than those in HTML ($p < 0.05$). In contrast, Python (17 LOC) and JavaScript (18 LOC) yield moderately sized outputs, with no significant difference between them ($p = 0.376$). Bash, Sh, and Shell produce the shortest snippets (3 LOC), with significantly lower LOC than all other languages (e.g., Bash vs. Python: $p < 0.05$). The LOC distributions reflect language-specific verbosity tendencies and align with prior findings [7], which report higher Source Lines of Code (SLOC) for C and C++ compared to Python and JavaScript.

LLM-generated code frequently includes multiple pro-

TABLE III: Top 5 pairwise co-occurrence statistics for LLM-generated programming languages. For each language pair (X, Y) , Count_XY indicates the number of conversations containing both X and Y .

Language{X}	Language{Y}	Count_XY	MLC-Rate
CSS	HTML	1,110	0.17
HTML	JavaScript	1,703	0.14
Java	XML	745	0.12
CSS	JavaScript	680	0.08
Bash	Python	2,057	0.07

gramming languages within the same conversation, with the highest MLC-Rates observed for CSS–HTML (0.17), HTML–JavaScript (0.14), and Java–XML (0.12). Among the 82,845 conversations, as shown in Table III, the most prevalent language combinations reflect typical programming workflows. For example, HTML and CSS are often combined for web content layout and styling, while HTML and JavaScript are used together to enable web page interactivity. Java–XML co-occurrence commonly arises in enterprise or Android development, where XML is used for configuration or user interface definitions alongside Java logic.

4) *Discussion:* In this research question, we analyze the conversational characteristics of developer–LLM interactions in CodeChat, which features informal, open-ended, and user-initiated conversations. This stands in contrast to the developer-task oriented workflows in DevGPT (i.e., pull requests and issue discussions on GitHub). An empirical study by Hao et al. [30] shows that multi-turn conversations account for 33.2% of pull request discussions and 36.9% of issue-related interactions in DevGPT [76]. Furthermore, Mondal et al. [40] identified *Missing Specifications*, *Additional Functionality*, and *Lack of Clarity in Generated Responses* as key prompt design gaps driving multi-turn interactions in DevGPT.

In contrast, our analysis shows that 68% of conversations in CodeChat are multi-turn, more than twice the rates reported in DevGPT. This indicates that iterative, back-and-forth interactions are even more prevalent in real-world, less structured programming environments. In CodeChat, *Different Use Cases*, *Missing Specifications*, and *Additional Functionality* are the most frequent gaps. The prominence of different use cases suggests that more explicit specification of intended use cases in initial prompts could help reduce unnecessary multi-turn interactions.

Summary of RQ1

LLM responses have a median *Token Ratio (TR)* of 14, reflecting substantial verbosity compared to developer prompts. $PL-Rate_X$ aligns with real-world trends for Python and C++, but diverges for JavaScript and Bash. High *MTC-Count* and frequent prompt design gaps indicate that iterative refinement is common.

B. RQ2: What topics do developers most commonly prompt when interacting with LLMs?

1) *Motivation:* Developers interact with LLMs to address a wide range of coding tasks, reflecting diverse intentions and tasks in their daily workflows. However, the specific topics developers prompt in actual practice remain largely unexplored. Understanding these topics is crucial for identifying the most pressing needs of developers and ensuring that LLMs provide relevant and effective support, and guiding tool integration into the appropriate workflows and IDE contexts. In this RQ, we aim to systematically uncover and categorize the common topics that developers prompt. These insights will help code assistant developers and researchers improve LLMs to better align with real-world programming tasks.

2) *Approach:* (a) *Topic Identification.* We leverage BERTopic [24], which is a topic modeling technique that combines transformer-based embeddings with clustering algorithms to generate interpretable topics from text data, to categorize common topics that developers inquire about in conversational scenarios. We conduct topic modeling on developer prompts as follows.

Step 1. Identify English conversations. Consistent with the BERTopic recommendation [25], which indicates that the default embedding model is designed for optimal performance on English text, we restrict our topic modeling to English prompts. CodeChat inherits language metadata from WildChat, allowing us to identify the language of each developer prompt. Using this metadata, we filter prompts by language and identify 52,086 English conversations, representing 62.9% of the full CodeChat dataset.

Step 2. Extract initial developer prompts. From these 52,086 English conversations, comprising 23,562 single-turn and 28,524 multi-turn conversations, we aim to characterize the initial intent topics that developers introduce when initiating interactions with LLMs. For single-turn conversations, the sole developer prompt is treated as the initial prompt, while for multi-turn conversations, we extract the developer’s first turn as the initial prompt.

Step 3. Topic modeling of initial intent. We apply BERTopic to model topics from the 52,086 initial English prompts. This step consists of two parts: (1) tuning BERTopic hyperparameters and (2) refining topic labels for interpretability. We encode the initial English developer prompts into vector representations using the “all-mpnet-base-v2” model from sentence transformers [57]. Next, we employ the Uniform Manifold Approximation and Projection (UMAP) [38], a dimensionality reduction technique commonly applied to high-dimensional datasets. UMAP utilizes manifold learning principles to preserve both global structure and local neighborhood relationships. After dimensionality reduction, we apply Hierarchical Density Based Spatial Clustering of Applications with Noise (HDBSCAN) [9], which is a robust clustering technique that identifies clusters of varying densities within the data.

To identify the optimal combination of hyperparameters, we conduct a grid search guided by the default and recommended settings from the BERTopic documentation [26]. Table IV lists

TABLE IV: Hyperparameter search space in topic modeling

Hyperparameters	Ranges	Description
n_neighbors	[5, 10, 15]	Controls how UMAP* balances local vs. global structure. Larger values create broader clusters.
n_components	[2, 5, 8]	Dimensionality of embeddings after reduction. Lower values simplify data but lose details.
min_cluster_size	[30, 40, ..., 200]	Minimum cluster size. Increasing this creates fewer, larger clusters.
min_samples	[5, 10, 15, 20]	Controls the number of outliers. Lower values allow more noise points.
top_n_words	[10]	Number of words per topic. Higher values might reduce coherence.
Total iterations	648	

*UMAP (Uniform Manifold Approximation and Projection): A dimensionality reduction technique that preserves both local and global data structure.

the hyperparameters and search ranges for exploring configurations that yield both meaningful and interpretable clusters. We select the optimal clustering configuration using a combination of quantitative metrics and qualitative interpretability.

From the grid search results, we select the top 10 hyperparameter combinations based on topic coherence scores [58], which evaluate the semantic interpretability and consistency of the generated topics by measuring the co-occurrence probabilities of the top words within each topic. To select the best-performing configuration, we manually examine the representative prompts generated by BERTopic for each of the top 10 configurations. Specifically, we manually assess whether representative prompts in each group share a coherent theme and align with the topic’s top keywords, discarding configurations with broad or inconsistent topics. We then retain one configuration that produces the most coherent and meaningful topics for further analysis.

To generate human-readable topic names, we combine topic keywords with representative developer prompts identified by BERTopic (i.e., those closest to each topic centroid). We then construct structured input prompts and use OpenAI’s GPT-4o [47] to produce concise and descriptive topic labels. To ensure quality, we conducted a manual evaluation of the generated topic labels, assessing their conciseness, semantic relevance, and interpretability based on a random sampled of 10 representative conversations and the key libraries referenced in LLM responses for each topic.

Step 4. Tracking intent shifts in multi-turn conversations. Using the initial intent topic model established on Step 3, we analyze the evolution of developer intent across 28,524 English multi-turn conversations. For each conversation, we assign an initial topic based on the developer’s first prompt. For each subsequent conversation turn, we combine all prior turn’s prompts and apply the trained BERTopic model to infer the topic distribution for the cumulative context. To detect intent shifts, we monitor the probability assigned to the initial topic across turns, defining a shift as a drop of more than 5% compared to the previous turn. Upon identifying

such shifts, we mark transitions to new topics, allowing us to trace the progression of developer intent throughout the conversation. Finally, we quantify engagement for each topic cluster by calculating the number of developer–LLM turns assigned to each topic across all 52,086 English conversations, capturing engagement patterns in both single-turn and multi-turn conversations.

(b) Topic Analysis. We use the per-turn topic assignments to further investigate how developers engage with different intent topics in LLM-based conversations. Our analysis consists of two steps: (1) examining the conversational engagement patterns associated with each topic, and (2) analyzing the distribution of programming languages across these topics.

Step 1. Examining engagement patterns across topics.

We begin by reporting the frequency of the top 10 intent topics derived from the topic model to provide an overview of the most common developer intents in 52,086 English developer–LLM conversations. We compute the turn count (TC) in each topic, which include both single-turn and multi-turn interactions. A higher TC for a topic cluster indicates more extensive interactions, which may reflect increased task complexity or the need for iterative clarification. We then employ the Scott–Knott test [61], which recursively partitions topics into clusters with similar mean turn counts (TC), to group topics with sufficiently similar engagement levels. Unlike pairwise statistical tests, Scott–Knott considers all topics simultaneously and produces an interpretable partitioning of engagement levels across all topics. The resulting clusters identify topics associated with higher or lower engagement levels, as measured by the number of conversational turns.

Step 2. Analyzing language distributions across topics.

To investigate programming language usage across topics, we extract language-tagged code snippets from LLM responses and analyze the code snippets associated with each topic cluster.

(c) Prompt Gap Sequence Analysis. To examine how prompt design gaps contribute to prolonged developer–LLM interactions, we analyze sequences of gap types assigned to consecutive developer prompt pairs, referred to as *chain-of-gap patterns*. A chain-of-gap pattern is defined as a sequence of prompt design gap labels occurring within a multi-turn conversation. Focusing on the topic cluster with the highest average turn count (as determined by the Scott–Knott test), we randomly sample 300 conversations based on a 95% confidence level and 5% margin of error. Each developer prompt pair in these conversations is labeled using our LLM-based annotation method (see Section IV-A2), resulting in a gap chain for each conversation.

To characterize prompt refinement strategies and recurring challenges, we extract all consecutive 3-gram sequences of gap types from each gap chain, allowing repeated gap types. Finally, we calculate the prevalence of the extracted 3-grams to identify recurring prompt design patterns that co-occur in extended interactions.

3) Findings: Web design and development is the most frequent topic, comprising 9.6% (4,995 out of 52,086) of conversations. As shown in Figure 7, developers seek

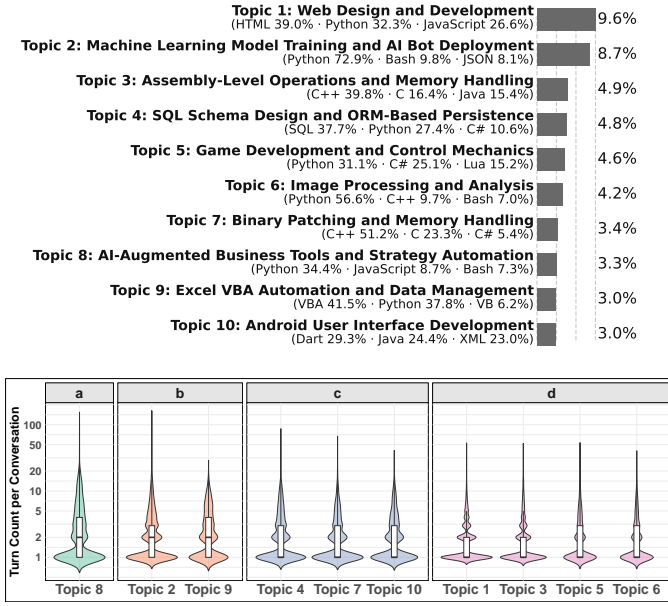


Fig. 7: (a) Top 10 most frequent topics in developer-LLM conversations; (b) Scott-Knott groupings for turn-count distributions per topic, reflecting the number of interactions across topics.



Fig. 8: Most frequent 3-gram "chain-of-gap" patterns in developer-LLM conversations within "Topic 8", the cluster with the highest average turn count. Each 3-gram represents a sequence of consecutive prompt design gaps.

assistance with front-end tasks such as layout design, responsive styling, and interactive components. This aligns with Anthropic’s analysis [2], which reports that coders commonly use AI for building user-facing applications, especially with web-development languages such as JavaScript and HTML, and that user interface tasks are among the top coding uses. Common keywords in this topic include `div`, used for structuring web pages in HTML [42], and `react`, a JavaScript library for building user interfaces [55]. This observation is also consistent with the findings in Section IV-A, where JavaScript-HTML and HTML-CSS are the most frequently co-occurring language pairs.

Machine learning model training and AI bot deployment are the second most frequent topic, comprising 8.7% (4,516 out of 52,086) of conversations. This topic

includes prompts where developers seek assistance with configuring training pipelines, such as implementing gradient scaling in PyTorch [50], designing deep learning architectures in Keras [12], and optimizing multi-task learning (MTL) workflows [16]. Prompts also address challenges related to deploying AI models, including inference request handling and output parsing when integrating model APIs.

Low-level programming and memory manipulation accounts for 8.3% (4,330 out of 52,086) of conversations, spanning assembly-level operations (4.9%) and binary patching (3.4%). The predominant languages in these topics are C++ and C, reflecting their central role in systems-level development. Developer prompts involve fine-grained memory control and low-level system behavior, with assembly instructions such as `mov`, `ptr`, and `eax`. In binary patching, developers seek assistance with runtime code modification, driver-level memory operations, and byte-level injection. These interactions highlight the application of LLMs in supporting tasks related to systems programming, reverse engineering, and low-level debugging.

Developer-LLM engagement differs significantly across topics as measured by TC, with four statistically distinct groups. As shown in Figure 7, Topic 8 (AI-augmented business tools and strategy automation) on its own forms one group (group “a”) identified by the Scott-Knott analysis, and shows the highest engagement with a mean of 3.40 turns. Group b includes Topic 2 (machine learning model training and AI bot deployment, mean = 2.89) and Topic 9 (Excel VBA automation and data management, mean = 2.86), both showing moderately high engagement. Group “c” contains Topic 4 (SQL schema design and ORM-based persistence, mean = 2.62), Topic 10 (Android user interface development, mean = 2.50), and Topic 7 (Binary patching and memory handling, mean = 2.42), reflecting moderate engagement levels. The remaining topics are in group “d”, which exhibits the lowest engagement levels.

In conversations on the topic with the highest TC (Topic 8), the most frequent prompt gap sequence involves three successive occurrences of Different Use Cases, accounting for 16.0% of the 681 observed 3-gram chains. As shown in Figure 8, the pattern involves repeated shifts in user intent or use cases, which is associated with higher turn counts. The next most common sequences are Additional Functionality → Additional Functionality → Different Use Cases (2.3%) and Different Use Cases → Different Use Cases → Additional Functionality (1.9%), both suggesting alternation between adding features and switching use cases before objectives are completed. We also observe Erroneous Response → Erroneous Response → Erroneous Response (1.9%), indicating consecutive LLM-generated incorrect outputs without effective correction and extend turn counts. These frequent gaps suggest that developers should carefully select use cases early to avoid repeated reframing, validate functionality before requesting new features to reduce unnecessary alternation, and incorporate explicit prompts for error recovery to limit cycles of repeated errors.

4) *Discussion:* Sagdic et al. [60] identify 17 distinct topics from developer-ChatGPT conversations by analyzing 1,710

prompts collected from GitHub. Their findings highlight specialized gaming-related conversations, such as *Web Game Implementation* (11.0%), and a broader focus on *Advanced Python Programming* (9.1%). Additionally, platform-centric topics like *GitHub and Git workflow* (8.1%) were prominent in their analysis. In contrast, our study examines 52,086 developer prompts from real-world conversations and analyzes the top 10 topics, which focus on general coding assistance. Specialized gaming-related conversations, such as *Game Development and Control Mechanics* (4.6%), are less prevalent in our findings. Instead, developers in our dataset focus on general webpage design tasks involving JavaScript and HTML. In comparison, Sagdic et al. [60] highlight the use of Node.js frameworks to enable server-side application logic and scalable systems. Although both studies indicate common Python-based tasks, our findings highlight machine learning model training and AI bot deployment (8.7%), involving frameworks like PyTorch and Keras.

Additionally, Sagdic et al. [60] identify the *DevOps integration and practices* topic (e.g., 8.1% of prompts are GitHub/Git workflows and 6.4% of prompts are Docker containerization) as prominent in DevGPT dataset, these infrastructure-focused topics are absent from our analysis. These differences underscore that DevGPT conversations shared on GitHub are more specialized and platform-oriented, whereas general developer-LLMs interactions in CodeChat tend to address broader software engineering practices.

Summary of RQ2

The analysis of 52,086 developer prompts reveals that developers frequently seek LLM assistance for *Web design and development* (9.6%), *Machine learning model training and AI bot deployment* (8.7%), and *Assembly-level operations and memory handling* (4.9%). Moreover, repeated shifts in use cases (16.0%) and alternation between features and objectives are prevalent prompt gaps in longer conversations.

C. RQ3: How high is the quality of LLM-generated code within coding conversations?

1) *Motivation:* As developers increasingly rely on LLMs to solve code-related tasks, the quality of LLM-generated code becomes a critical concern. While LLMs have demonstrated impressive capabilities, the generated code may contain defects or fail to follow best practices. A comprehensive evaluation of LLM-generated code across different programming languages and real-world scenarios can provide valuable insights into the strengths and weaknesses. The findings of this RQ can help developers make more informed and responsible usage of LLMs, guide code assistant developers to address identified issues, and offer research directions for improving the capabilities and reliability of LLMs in code-related tasks.

2) *Approach:* (a) *Identifying conversations in top 5 programming languages.* Using the $PL\text{-}Rate_X$ results from RQ1 IV-A, we select the five most frequently generated programming languages. We exclude Bash due to the lack

TABLE V: Summary of conversations for RQ3 and code snippet statistics in the top five programming languages.

Programming Language	#Convo	STC-Count	MTC-Count	#first-turn Code snippets
Python	22,539	11,538	11,001	32,586
JavaScript	7,385	4,353	3,032	10,128
C++	5,404	2,780	2,624	7,788
Java	4,557	2,496	2,061	6,922
C#	4,355	2,328	2,027	6,261
Total	44,240	23,495	20,745	63,685

*Conversations (abbreviated as Convo in the table); STC-Count is the number of single-turn conversations; MTC-Count is the number of multi-turn conversations;

of robust static analysis tools for it and due to the limited structural complexity of typical Bash code. This yields 44,240 conversations with Python, JavaScript, C++, Java, or C# code snippets. From each conversation, we extract the first-turn code snippet to assess the quality of the initial LLM-generated response and classify the entire conversation as single-turn or multi-turn based on its turn count (TC). Table V presents the distribution of conversations and their corresponding first-turn code snippets by programming language, with multi-turn conversations retained for further analysis of code evolution.

(b) *Detecting code clones across turns in multi-turn conversations.* From the 20,745 multi-turn conversations, we analyze how code quality issues evolve across the outputs of consecutive turns within the same conversation. As our findings in IV-B indicate that unrelated tasks are common in multi-turn conversations, we filter out all subsequent conversational turns where developers switch tasks or when the LLM generates unrelated code. To detect such task switches, we apply code clone detection by checking the similarity between code snippets from two adjacent turns. If two successive conversation turns yield sufficiently similar code, we consider both turns to be related and add the second turn to the current task sequence. A **task sequence** is defined as a consecutive set of turns where the generated code remains sufficiently similar, reflecting work on the same programming task. However, if the similarity between both turns' code is too low, we consider that a new task has started, ending the previous task sequence.

We employ C4 [68], a contrastive learning-based model for detecting near-miss (i.e., code snippets that maintain structural similarity despite modifications such as statement addition, removal, or alteration) and semantic code clones (i.e., code snippets that achieve the same functional behaviour despite different syntactic or structural implementations) across programming languages. C4 compares learned code representations to identify consecutive outputs that are structurally or functionally related, indicating contextual continuity rather than a shift to a new, unrelated prompt. We apply C4 to multi-turn conversations involving the five most frequent programming languages (Table V), using the configuration from [68] that achieved the highest precision and recall on the reference dataset. Table VI presents the resulting counts of follow-up

TABLE VI: Count of developer-LLM conversations with follow-up multi-turn interactions through C4, a contrastive learning-based code clone detection model

Language (MTC-Count)	2-turn Convo	3-turn Convo	4-turn Convo	5-turn Convo	5+ turn Convo	% of MTC (After C4)
Python (11,001)	3,299	1,487	937	624	2,519	80.6%
JavaScript (3,032)	947	403	245	174	592	77.9%
C++ (2,624)	674	455	268	180	485	78.6%
Java (2,061)	597	269	178	91	457	77.2%
C# (2,027)	628	287	183	128	372	78.8%
Total (20,745)	6,145	2,901	1,811	1,197	4,425	79.4%

*MTC (abbreviated as Multi-turn Conversations in the table);

multi-turn conversations identified across the entire dataset for top 5 programming languages.

(c) *Assessing the evolution of code quality of LLM generated code.* To evaluate the evolution of LLM-generated code quality, we use established code quality tools to detect potential defects (e.g., undefined variables, null pointer dereferences) and code smells (e.g., stylistic issues) across the different code versions in the analyzed task sequences. For Python and Java, we follow the approach in Liu et al. [37], which leverages Pylint [52] for Python and PMD [51] for Java. For JavaScript, we use ESLint [20], which is a widely adopted tool for detecting syntax errors and style violations. For C++, we employ Cppcheck [15], which captures a broad range of issues, including performance and portability concerns. For C#, we utilize Roslyn [39], which is integrated into the .NET SDK for inspecting syntax, semantics, and maintainability issues.

For each programming language, we identify the most frequent quality issue types detected in the first-turn response and track their prevalence across subsequent turns to reveal fine-grained trends in quality issue evolution. Specifically, we compute the percentage of task sequences containing at least one instance of quality issue type m in turn N as $P_{m,N} = C_{m,N}/C_N \times 100$, where $C_{m,N}$ is the number of task sequences with at least one occurrence of m in turn N , and C_N is the total number of task sequences with at least N turns.

To evaluate whether the frequency of specific quality issue types changes across conversational turns, we conduct trend analysis on the computed $P_{m,N}$ values for each issue type. Specifically, we perform linear regression[41], modeling $P_{m,N}$ as a function of turn number N . A statistically significant regression slope ($p < 0.05$) is interpreted as evidence of a trend. In cases where the data may violate linearity or normality assumptions, we apply nonparametric Mann-Kendall trend test[29], which is robust to non-normal distributions and outliers. This combined methodology ensures the reliability and validity of our assessment of issue prevalence trends across multi-turn interactions.

TABLE VII: Static analysis results of the first-turn LLM-generated code snippets

	Severity	#Code Snippets	%Code Snippets	Top3_Msg
Python	Error	10,037	30.8%	E0602:UndefinedVariable
		6,767	20.8%	E0401:ImportError
		2,617	8.0%	E0001:SyntaxError
	Warning	3,236	9.9%	W0621:RedefinedOuterName
		3,079	9.4%	W0611:UnusedImport
		1,214	3.7%	W0612:UnusedVariable
	Refactor	1,351	4.2%	R0903:TooFewPublicMethods
		861	2.6%	R1705:NoElseReturn
		451	1.4%	R0917:TooManyPositionalArgs
	Convention	27,174	83.4%	C0103:InvalidName
10,474		32.1%	C0116:MissingFunctionDocs	
5,179		15.9%	C0301:LineTooLong	
JS	Error	7,627	75.3%	NoUndef
		1,460	14.4%	SyntaxError
	Warning	3,413	33.7%	NoUnusedVars
C++	Error	733	9.4%	SyntaxError
	Warning	26	0.3%	NoOperatorEq
	Performance	99	1.3%	FunctionStatic
		84	1.1%	PassedByValue
	Style	1,382	17.8%	UnusedFunction
		199	2.6%	NoExplicitConstructor
		175	2.2%	UnreadVariable
	Information	3,198	41.1%	MissingIncludeSystem
		517	6.6%	MissingInclude
	Java	Critical	131	1.9%
82			1.2%	Style:FieldNaming
74			1.1%	Style:LocalVariableNaming
High		1,552	22.4%	BestPractices:SystemPrintln
Medium		5,253	75.9%	Documents:CommentRequired
		3,291	47.5%	Design:UseUtilityClass
		3,139	45.3%	Style:LocalVarCouldBeFinal
Low		567	8.2%	Style:ShortClassName
		366	5.3%	Style:UnnecessaryImport
		110	1.6%	BestPractices:UseVarArgs
C#	Error	3,079	49.2%	CS0246:NamespaceUnfound
		1,522	24.3%	CS0103:NameIdentifierNotExist
		1,107	17.7%	CS0106:ModifierNotValid
	Warning	2,652	42.4%	EnableGenDocFile
		600	9.6%	CA1852:SealInternalTypes
		328	5.2%	CA1050:TypesInNamespaces
	Style	623	10.0%	IDE0040:AddAccessModifiers
		612	9.8%	IDE0055:FormattingRule
506		8.1%	IDE0210:ToTopLevelStatements	

(d) *Analyzing prompts that resolve syntax errors.* To understand how prompts help to resolve syntax errors, we also analyze conversational prompt types associated with resolving syntax errors in multi-turn conversations. The syntax error quality issue type is chosen because it represents fundamental correctness issues that prevent code execution and can be reliably detected by static analysis. From CodeChat, we extract 788 multi-turn conversations with Python, JavaScript, or C++ code snippets that include at least one pair of adjacent LLM responses in which the first contains at least one syntax error and the second contains none. To ensure statistical validity, we randomly sample 267 cases from these conversations using the standard estimation method for large populations at a 95% confidence level and a 5% margin of error.

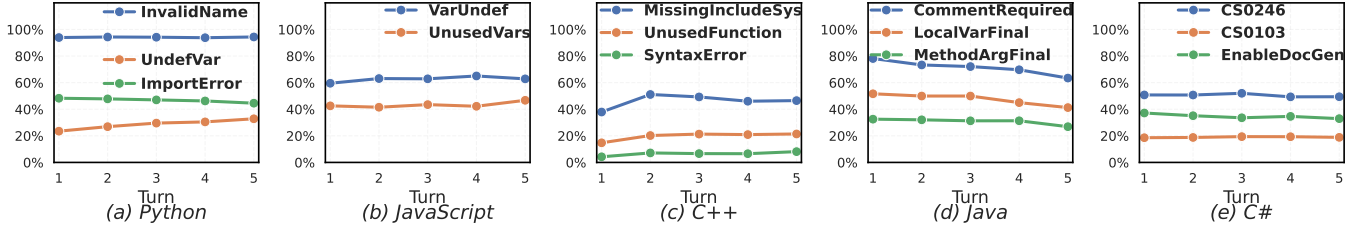


Fig. 9: Distribution of the top three quality issues across turns in multi-turn conversations, ranked by their frequency at the first turn.

Labeling is performed independently by the first author and a fifth-year Ph.D. student in software engineering, following Shin et al. [62]’s nine-category scheme (as shown in Table VIII) of conversational prompts for guiding LLMs in code generation tasks. Cases that do not match any category are jointly reviewed, and we inductively define two additional categories: *Change Use Cases*, where the developer redirects the LLM to a different coding goal, and *Request Full Code*, where the developer asks for the complete corrected program rather than partial fixes. Inter-annotator agreement for all categories is computed using Cohen’s kappa and equals 0.90, indicating almost perfect agreement. Remaining disagreements are resolved through discussion until consensus is reached.

3) *Findings*: In Python, invalid naming is the most frequent quality issue, affecting 83.4% of LLM-generated snippets; across multi-turn conversations, the proportion of undefined-variable issues increases, while the proportion of import-error issues decreases. As shown in Table VII, 83.4% of the first-turn Python snippets contain naming issues that fail to comply with Pylint’s default PEP8 rules [71]. Undefined variables (E0602) occur in 46.7% of error-type issues (30.8% of snippets) and unused imports (W0611) in 22.8% of warning-type issues (9.4% of snippets). In human-written Python code of Jupyter notebooks [63], undefined variables occur in 46.3% of error-type issues and unused imports in 16.7%, suggesting that LLMs mirror human patterns for undefined variables but introduce unused imports more often. Syntax errors appear in 8.0% of the first-turn snippets, less than the 26.0% in human-written Jupyter notebook code [78], but their presence can still halt code execution and necessitate manual correction.

In follow-up multi-turn Python conversations (Figure 9), the prevalence of *UndefinedVar* increases from 23.5% of code snippets at turn 1 to 32.8% at turn 5 ($p < 0.05$), indicating challenges in preserving variable context, while the proportion of *ImportError* decreases from 48.3% of code snippets to 44.6% ($p < 0.05$), suggesting missing imports are progressively added.

In JavaScript, undefined variables are the most frequent quality issue, affecting 75.3% of LLM-generated snippets; unused variables occur in 33.7%, and syntax errors in 14.4%, with no significant changes across turns. As shown in Table VII, undefined variables appear in 7,627 of the 10,126 first-turn JavaScript snippets (75.3%), which is lower than the 91.0% of variable-related violations reported for human-

written JavaScript on Stack Overflow [22]. Syntax errors are found in 1,456 snippets (14.4%), substantially higher than the 1.9% syntax error rate in human-written Stack Overflow code [22]. In LLM-generated JavaScript, the most frequent cause of these syntax errors are unexpected angle brackets, which occur in 5.8% of code snippets. In multi-turn JavaScript conversations (Figure 9), the prevalence of top issues remains statistically unchanged ($p > 0.05$).

In C++, missing `#include` headers are the most frequent quality issue, affecting 41.1% of initial-turn LLM-generated snippets; unused functions occur in 17.8%, and syntax errors in 9.4%, with no significant changes across turns. As shown in Table VII, syntax errors are present in 733 snippets (9.4%), which is less than the 20.0% reported for human-written C/C++ code on Stack Overflow when analyzed with CppCheck [79].

In follow-up multi-turn C++ conversations (Figure 9), the prevalence of *MissingIncludeSystem* rises from 37.9% of code snippets at turn 1 to 46.5% at turn 5, and *SyntaxError* from 4.2% to 8.1%, but neither change is statistically significant ($p > 0.05$).

In Java, missing required comments are the most frequent quality issue, affecting 75.9% of initial-turn LLM-generated snippets, with documentation violations and missed opportunities to declare local variables as `final` both declining across turns. As shown in Table VII, absence of `final` on local variables (*LocalVariableCouldBeFinal*) occurs in 3,135 snippets (45.3%) and is consistent with human-written Java [36], where such violations appear in 41.1% of pull requests from 28 large projects.

In follow-up multi-turn Java conversations (Figure 9), the prevalence of *CommentRequired* prevalence decreases from 78.1% of code snippets at turn 1 to 63.4% at turn 5 ($p < 0.05$), and *LocalVariableCouldBeFinal* from 51.6% to 41.3% ($p < 0.05$), suggesting iterative improvements in documentation and in declaring local variables as `final`.

Finally, in C#, unresolved namespaces are the most frequent quality issue, affecting 49.2% of initial-turn LLM-generated snippets. Across multi-turn conversations, their prevalence remains unchanged ($p > 0.05$). As shown in Table VII, missing documentation occurs in 42.4% of code snippets (2,652 out of 6,261), which can hinder code comprehension and maintenance. Other notable issues include omitted accessibility modifiers (10.0%) and formatting violations (9.8%), both of which impact code readability. Compared

TABLE VIII: Categories of follow-up prompts that resolve syntax errors, adapted from Shin et al. [62]

Category Name	Frequency	Percentage
Point out mistake then request fix	61	22.8%
Ask questions to guide correct solution	45	16.9%
Add specific instructions	44	16.5%
Request improvements	37	13.9%
Request Full Code*	29	10.9%
Add more context	19	7.1%
Request more description about code	12	4.5%
Change Use Cases*	8	3.0%
Request examples	6	2.2%
Request alternative code generation	3	1.1%
Request verification	3	1.1%
Total	267	100.0%

* Categories defined in this study.

to human-written C# code in 100 GitHub projects analyzed by [45], accessibility omissions (members that could be made private) occur in 5.4% of analyzed cases, and formatting inconsistencies occur in 3.33%, suggesting that LLM-generated code mirrors these human patterns but at higher frequencies.

“Point mistake out then request fix” is the most common prompt type for resolving syntax errors, occurring in 22.8% of the 267 sampled resolution cases. Explicitly naming the error and requesting a fix focuses the LLMs on the faulty code and is associated with syntactically valid code generation. The next most frequent prompt types are *Ask questions to guide correct solution* (16.9%) and *Add specific instructions* (16.5%), both of which guide the LLMs toward correct syntax by clarifying intent or specifying concrete edits. Together, the three prompt types account for 56.2% of the resolved cases, suggesting that clear error signaling, guided questioning, and precise directives are prevalent in successful resolutions and may reduce turn count when seeking non-syntax-error outputs.

4) *Discussion:* Liu et al. [37] study ChatGPT-generated code in Python and Java for 2,033 programming tasks from LeetCode, identifying style and maintainability issues. While their analysis focuses on two languages in a controlled problem-solving setting (i.e., LeetCode), our study investigates a considerably larger dataset consisting of 63,685 code snippets generated by ChatGPT across five languages: Python, Java, JavaScript, C++, and C#.

Our findings confirm prior observations regarding issues in Python and Java [37], while also uncovering additional issues. In Python, we identify quality issues like variables being declared but not used appearing at a proportion of 3.7%, and the absence of an else in return statements at 2.6%, compared to the prior findings of 5.1% and 7.9%, respectively, reported by Liu et al. [37]. Our findings further reveal broader challenges with variable definitions and import errors, affecting 30.8% and 20.8% of the code snippets. These results suggest deeper syntax and dependency management issues requiring attention. In Java, Liu et al. [37] report that the most common issues in ChatGPT-generated code for LeetCode problems are multiple variable declarations (16.4%, where

each variable should be declared in a separate statement) and parameter reassignment (8.7%, where method parameters are reassigned within the method body). In contrast, we identify significant documentation and design shortcomings across general-purpose code generation tasks. For example, 75.9% of code snippets show inadequate commenting, and 47.5% lack proper utility class structures, highlighting critical areas that need focused improvement to enhance overall code quality.

Summary of RQ3

The most frequent issues in LLM-generated code are: invalid naming in Python (83.4% of code snippets), undefined variables in JavaScript (75.3% of code snippets), missing `#include` in C++ (41.1%), missing required comments in Java (75.9%), and namespace unfound errors in C# (49.2%). Moreover, in multi-turn conversations, Python import errors decrease from 48.3% at turn 1 to 44.6% at turn 5, while Java documentation violations drop from 78.1% to 63.4%.

V. IMPLICATIONS

In this section, we discuss the implications of our findings for developers using LLMs, conversational code assistants, and researchers in software engineering.

A. Implications for developers Using LLMs

Developers should employ structured post-response verification workflows and careful judgment processes before integrating LLM-generated code into projects. Section IV-C shows that LLM-generated code often contains syntax errors (e.g., 8.0% of Python snippets, 14.4% of JavaScript snippets), structural problems (e.g., unused code segments appearing in 33.7% of JavaScript snippets), and maintainability concerns (e.g., invalid naming conventions found in 83.4% of Python snippets). These issues can compromise code readability, increase technical complexity, and hinder long-term maintenance. To mitigate such risks, developers should implement a structured post-response verification workflow, which involves systematically reviewing LLM-generated code using static analysis tools and linters, interpreting diagnostic feedback, and iteratively refining prompts to resolve detected issues.

B. Implications for Conversational Code Assistants

Conversational code assistants should support artifact management. LLM-generated code from chat-based assistants such as ChatGPT and Copilot Chat is typically represented as textual strings, often consisting of multiple snippets that belong to different files or span across different languages (Section IV-A). This limitation is not specific to WildChat, but reflects the current landscape, where artifact management is rarely supported. Conversational code assistants should be capable of managing these code fragments as artifacts that can be version-controlled, organized, and executed when self-contained with the necessary dependencies. For instance, some platforms [3] automatically save LLM-generated code into

separate files (artifacts), allowing users and code assistants to track, edit, execute, or share these artifacts more effectively than working with isolated code snippets.

Conversational code assistants should introduce automated post-generation workflows by embedding code-checking and correction tools into their response pipelines, rather than relying on developers’ manual checks or external utilities. Section IV-C reveals frequent syntax errors and style violations. Existing solutions, such as internal iterative feedback loops [32], code formatters like Python’s Black [53], and tools provided by Model Context Protocol (MCP) servers [31], help address surface-level issues but cannot fully resolve deeper syntax or logic errors and still require significant human oversight. We recommend conversational code assistants extend to this inner-loop feedback approach into post-generation workflows, applying static analysis, syntax correction, and formatting tools automatically after generating initial LLM outputs but before presenting code to developers.

C. Implications for IDE Tool Builders

IDE tool builders should integrate conversational interactions with automated multi-language context management into IDE workflows. Section IV-A shows that developers frequently request LLM-generated code snippets that span multiple source files and programming languages (e.g., Bash-Python, HTML-JavaScript). Emerging tools such as Cursor [6] and Cline [13] embed conversational interfaces into IDEs, enabling multi-language code generation, explanation, and refactoring. However, current workflows depend heavily on manual steps such as copy-pasting or prompt-based interactions, with limited support for direct snippet insertion like drag-and-drop. While these tools can reference multiple files and offer language-aware suggestions, features such as automated dependency tracking, cross-file linking, and seamless multi-language handling remain limited. To improve integration, IDEs should support intuitive insertion methods and automatically manage dependencies, synchronize updates across files, and apply language-specific templates.

IDE tool builders should support context-aware prompting and version management for iterative conversational interactions. As shown in Sections IV-A and IV-B, developers frequently seek LLM assistance across diverse and specialized tasks such as AI-Augmented Business Tools and Strategy Automation, and Machine Learning Model Training and AI Bot Deployment, often requiring multiple exploratory conversational turns. To enhance developers’ efficiency and workflow continuity, IDEs should offer proactive prompting features by automatically detecting the user’s context (e.g., focused file, project dependencies, frameworks in use) and suggesting contextually relevant conversational prompts. Additionally, IDEs should natively incorporate automated versioning, rollback, and branching capabilities specifically tailored for LLM-generated artifacts, allowing developers to conveniently navigate iterative dialogue states, experiment safely, and manage generated solutions with greater confidence and ease.

D. Implications for Researchers

Researchers should investigate dynamic token allocation and optimized code tokenization strategies to address accuracy and cost-quality trade-off in LLM-generated code. As discussed in Section IV-A, LLM responses in developer-LLM conversations are 2.4 times longer than typical human-written answers on Stack Overflow. These verbose outputs inflate token-generation costs and restrict multi-turn interactions within limited context windows. Moreover, in Section IV-C, the lengthy outputs frequently show quality issues. Researchers should optimize token usage through techniques such as prompt learning and attention mechanisms (e.g., CodePrompt [11]) and token prioritization strategies (e.g., dynamic scoring code token trees [54]). By tackling prompt-length variability and optimizing token allocation, researchers can deliver more concise responses, enhance the scalability and the cost-quality trade-off, reduce computational costs, and ultimately boost developer productivity.

Researchers should propose new benchmarks to evaluate LLMs’ performance in tasks that users frequently seek assistance with. As highlighted in Section IV-B, the results show that Web design and development is the most common topic (9.6% of prompts), followed by machine learning model training and AI bot deployment (8.7%). Other prominent domains include SQL schema design, game development and image processing and analysis. While benchmarks like HumanEval-X [82] evaluate LLMs’ ability to generate functionally correct code and support tasks like code translation, they may not fully cover the diverse needs of developers in real-world scenarios. Researchers should create benchmarks that reflect real-world tasks by referencing the frequent topics from developer prompts from our paper to ensure LLMs can effectively assist users in these critical domains.

Researchers should enhance LLMs’ ability to generate comprehensive and maintainable code documentation. The findings in Section IV-C reveal gaps in LLM-generated documentation, for example, 32.1% of Python snippets are lacking function-level documentation. Researchers should develop methods to improve LLMs’ ability to generate context-aware documentation, including practical guidance, usage examples, and automated updates in response to code changes. This would enhance the usability and maintainability of LLM-generated code.

VI. THREATS TO VALIDITY

A. Internal Validity

We select static analysis tools that are widely recognized for their reliability and relevance to the programming languages under investigation. However, a potential internal validity threat arises from the specific configuration settings used for each static analysis tool. To mitigate this concern, we follow the official documentation and recommended guidelines for each linter, thereby ensuring consistency, reliability, and transparency throughout our evaluation process.

Another limitation is that our code quality assessment is restricted to the types of issues supported by static analyzers. Other quality attributes, such as code complexity, security

vulnerabilities, are beyond the scope of our current analysis and are therefore not reflected in our findings.

Finally, a potential threat arises from the manual labeling of conversational prompts for syntax error resolution, which is inherently subjective. Although two software engineering researchers independently labeled the data and resolved disagreements through discussion, there remains a risk of bias or misclassification, particularly given the limited number of annotators. Nevertheless, the high agreement between both annotators (Cohen's kappa = 0.90) suggests strong reliability in the labeling process.

B. External Validity

Since CodeChat focuses on GPT-based models, our results may not extend to other LLMs. Additionally, though CodeChat is a large-scale dataset of developer-LLM conversations, it may not fully represent all software development contexts. Future work could expand this research by incorporating interactions from other LLMs and datasets, and extending the topic analysis in Section IV-B to non-English prompts.

VII. CONCLUSION

The widespread adoption of Large Language Models (LLMs) by developers has transformed software engineering workflows. However, the lack of real-world conversational data limits the understanding of these interactions. This study bridges this gap by analyzing developer-LLM interactions, identifying common coding tasks, and assessing the quality of LLM-generated code in practical scenarios. We introduce CodeChat, a filtered subset of the WildChat dataset, containing 82,845 real-world developer-LLM conversations with 368,506 code snippets across over 20 programming languages.

Our findings reveal that developers rely on LLMs for tasks such as webpage design, machine learning model training, and low-level memory operations. LLM responses often include detailed, multi-language code snippets that exceed the length of developer prompts. However, LLM-generated code frequently contains defects, including syntax issues, undefined variables, and maintainability problems.

In the future, we aim to develop domain-specific benchmarks targeting the most frequently requested programming tasks and to design advanced error-correction techniques that leverage static analysis feedback, with the goal of improving the usability and trustworthiness of LLM-generated code.

ACKNOWLEDGMENT

We would like to thank Fangjian Lei at Queen's University for his valuable assistance in manually validating our results. His contributions are essential in confirming the reliability and precision of our findings.

REFERENCES

- [1] S. Almanasa and K. Suwais, "Analysis of ChatGPT-Generated Codes Across Multiple Programming Languages," *IEEE Access*, vol. 13, pp. 23 580–23 596, 2025.
- [2] Anthropic, "Anthropic economic index: Ai's impact on software development," <https://www.anthropic.com/research/impact-software-development>, 2025, accessed: 2025-09-01.
- [3] Anthropic, "Artifacts are now generally available," <https://www.anthropic.com/news/artifacts>, 2025, accessed: 2025-09-01.
- [4] Anthropic, "Claude ai: Next-generation ai assistant," <https://claude.ai/>, 2025, accessed: 2025-09-01.
- [5] Anthropic, "Privacy policy," <https://www.anthropic.com/legal/privacy>, 2025, accessed: 2025-09-01.
- [6] Anysphere Inc., "Cursor: The ai code editor," <https://cursor.com/en>, 2025, accessed: 2025-09-01.
- [7] L. Ardito, L. Barbato, R. Coppola, and M. Valsesia, "Evaluation of rust code verbosity, understandability and complexity," *PeerJ Computer Science*, vol. 7, p. e406, 2021.
- [8] Author or Organization, "Title or description of the website," <https://x.com/>, 2025, accessed: 2025-09-01.
- [9] R. J. G. B. Campello, D. Moulavi, and J. Sander, "Density-Based Clustering Based on Hierarchical Density Estimates," in *Advances in Knowledge Discovery and Data Mining*, J. Pei, V. S. Tseng, L. Cao, H. Motoda, and G. Xu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 160–172.
- [10] W.-L. Chiang, Z. Li, Z. Lin, Y. Sheng, Z. Wu, H. Zhang, L. Zheng, S. Zhuang, Y. Zhuang, J. E. Gonzalez, I. Stoica, and E. P. Xing, "Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality," March 2023.
- [11] Y. Choi and J.-H. Lee, "CodePrompt: Task-Agnostic Prefix Tuning for Program and Language Generation," in *Findings of the Association for Computational Linguistics: ACL 2023*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, jul 2023, pp. 5282–5297.
- [12] F. Chollet, "Keras," <https://github.com/keras-team/keras>, 2015, accessed: 2025-09-01.
- [13] Cline Bot Inc., "Cline: Ai coding, open source and uncompromised," <https://cline.bot/>, 2025, accessed: 2025-09-01.
- [14] T. Coignon, C. Quinton, and R. Rouvoy, "A Performance Study of LLM-Generated Code on Leetcode," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 79–89.
- [15] Cppcheck Team, "Cppcheck: A static analyzer for c/c++ code," <https://cppcheck.sourceforge.io/>, 2025, accessed: 2025-09-01.
- [16] M. Crawshaw, "Multi-task learning with deep neural networks: A survey," *arXiv preprint arXiv:2009.09796*, 2020. [Online]. Available: <https://arxiv.org/abs/2009.09796>
- [17] J. K. Das, S. Mondal, and C. K. Roy, "Why Do Developers Engage with ChatGPT in Issue-Tracker? Investigating Usage and Reliance on ChatGPT-Generated Code," *arXiv preprint arXiv:2412.06757*, 2024.
- [18] Z. Delile, S. Radel, J. Godinez, G. Engstrom, T. Brucker, K. Young, and S. Ghanavati, "Evaluating Privacy Questions from Stack Overflow: Can ChatGPT Compete?" in *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*, 2023, pp. 239–244.
- [19] A. Dinno, "Nonparametric pairwise multiple comparisons in independent groups using dunn's test," *The Stata Journal*, vol. 15, no. 1, pp. 292–300, 2015.
- [20] ESLint Team, "Getting started with eslint - pluggable javascript linter," <https://eslint.org/docs/latest/use/getting-started>, 2025, accessed: 2025-09-01.
- [21] Y. Feng, S. Vanam, M. Cherukupally, W. Zheng, M. Qiu, and H. Chen, "Investigating Code Generation Performance of ChatGPT with Crowdsourcing Social Data," in *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2023, pp. 876–885.
- [22] U. Ferreira Campos, G. Smethurst, J. P. Moraes, R. Bonifácio, and G. Pinto, "Mining rule violations in javascript code snippets," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 195–199.
- [23] GitHub, Inc., "Github: Build and ship software on a single, collaborative platform," <https://github.com/>, 2025, accessed: 2025-09-01.
- [24] M. Grootendorst, "Bertopic: Neural topic modeling with a class-based tf-idf procedure," *arXiv preprint arXiv:2203.05794*, 2022, <https://maartengr.github.io/BERTopic/index.html> Accessed: 2025-09-01.
- [25] M. Grootendorst, "Bertopic frequently asked questions," <https://maartengr.github.io/BERTopic/faq.html>, 2025, accessed: 2025-09-01.
- [26] M. Grootendorst, "Bertopic parameter tuning guide," https://maartengr.github.io/BERTopic/getting_started/parameter%20tuning/parametertuning.html, 2025, accessed: 2025-09-01.
- [27] X. Gu, M. Chen, Y. Lin, Y. Hu, H. Zhang, C. Wan, Z. Wei, Y. Xu, and J. Wang, "On the Effectiveness of Large Language Models in Domain-Specific Code Generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 3, feb 2025.

- [28] M. A. Haider, A. B. Mostofa, S. S. B. Mosaddek, A. Iqbal, and T. Ahmed, "Prompting and Fine-tuning Large Language Models for Automated Code Review Comment Generation," *arXiv preprint arXiv:2411.10129*, 2024.
- [29] K. H. Hamed and A. R. Rao, "A modified mann-kendall trend test for autocorrelated data," *Journal of Hydrology*, vol. 204, no. 1, pp. 182–196, 1998.
- [30] H. Hao, K. A. Hasan, H. Qin, M. Macedo, Y. Tian, S. H. Ding, and A. E. Hassan, "An empirical study on developers' shared conversations with ChatGPT in GitHub pull requests and issues," *Empirical Software Engineering*, vol. 29, no. 6, p. 150, 2024.
- [31] M. M. Hasan, H. Li, E. Fallahzadeh, G. K. Rajbahadur, B. Adams, and A. E. Hassan, "Model context protocol (mcp) at first glance: Studying the security and maintainability of mcp servers," 2025.
- [32] D. Huang, Q. Bu, Y. Qing, and H. Cui, "CodeCoT: Tackling Code Syntax Errors in CoT Reasoning for Code Generation," *arXiv preprint arXiv:2308.08784*, 2024.
- [33] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A Survey on Large Language Models for Code Generation," *arXiv preprint arXiv:2406.00515*, 2024.
- [34] S. Kabir, D. N. Udo-Imeh, B. Kou, and T. Zhang, "Who Answers It Better? An In-Depth Analysis of ChatGPT and Stack Overflow Answers to Software Engineering Questions," *CoRR*, vol. abs/2308.02312, 2023.
- [35] A. Köpf, Y. Kilcher, D. von Rütte, S. Anagnostidis, Z. R. Tam, K. Stevens, A. Barhoum, D. Nguyen, O. Stanley, R. Nagyi *et al.*, "Openassistant conversations-democratizing large language model alignment," *Advances in Neural Information Processing Systems*, vol. 36, 2024.
- [36] V. Lenarduzzi, V. Nikkola, N. Saarimäki, and D. Taibi, "Does code quality affect pull request acceptance? an empirical study," *Journal of Systems and Software*, vol. 171, p. 110806, 2021.
- [37] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B. D. Le, and D. Lo, "Refining chatgpt-generated code: Characterizing and mitigating code quality issues," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 5, pp. 1–26, 2024.
- [38] L. McInnes, J. Healy, and J. Melville, "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction," *arXiv preprint arXiv:1802.03426*, 2020.
- [39] Microsoft, "Roslyn analyzers github repository," <https://github.com/dotnet/roslyn-analyzers>, 2025, accessed: 2025-09-01.
- [40] S. Mondal, S. D. Bappon, and C. K. Roy, "Enhancing user interaction in chatgpt: Characterizing and consolidating multiple prompts for issue resolution," in *Proceedings of the 21st International Conference on Mining Software Repositories*, ser. MSR '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 222–226.
- [41] D. C. Montgomery, E. A. Peck, and G. G. Vining, *Introduction to linear regression analysis*. John Wiley & Sons, 2021.
- [42] Mozilla Contributors, "The content division element," <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/div>, 2025, accessed: 2025-09-01.
- [43] M. Nayeibi and B. Adams, "Image-based communication on social coding platforms," *J. Softw. Evol. Process*, vol. 36, no. 5, Apr. 2024.
- [44] N. Nikolaidis, K. Flamos, K. Gulati, D. Feitosa, A. Ampatzoglou, and A. Chatzigeorgiou, "A Comparison of the Effectiveness of ChatGPT and Co-Pilot for Generating Quality Python Code Solutions," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)*, 2024, pp. 93–101.
- [45] M. Odermatt, D. Marcilio, and C. A. Furia, "Static analysis warnings and automatic fixing: A replication for C# projects," in *Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 805–816.
- [46] OpenAI, "Chatgpt," <https://chatgpt.com/>, 2025, accessed: 2025-09-01.
- [47] OpenAI, "Hello gpt-4o," <https://openai.com/index/hello-gpt-4o>, 2025, accessed: 2025-09-01.
- [48] OpenAI, "Privacy policy," <https://openai.com/policies/privacy-policy>, 2025, accessed: 2025-09-01.
- [49] OpenAI, "tiktoken: A fast bpe tokeniser for use with openai's models," <https://github.com/openai/tiktoken>, 2025, accessed: 2025-09-01.
- [50] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems* 32. Curran Associates, Inc., 2019, pp. 8024–8035.
- [51] PMD Team, "Pmd - source code analyzer documentation," <https://pmd.github.io/pmd/index.html>, 2025, accessed: 2025-09-01.
- [52] Pylint Development Team, "Pylint tutorial and documentation," <https://pylint.readthedocs.io/en/stable/tutorial.html>, 2025, accessed: 2025-09-01.
- [53] Python Software Foundation, "Black: The uncompromising python code formatter," <https://github.com/psf/black>, 2025, accessed: 2025-09-01.
- [54] M. Qu, J. Liu, L. Kang, S. Wang, D. Ye, and T. Huang, "Dynamic Scoring Code Token Tree: A Novel Decoding Strategy for Generating High-Performance Code," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1308–1318.
- [55] React Team, "React - the library for web and native user interfaces," <https://react.dev>, 2025, accessed: 2025-09-01.
- [56] Reddit Inc., "Reddit," <https://www.reddit.com/>, 2025, accessed: 2025-09-01.
- [57] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, K. Inui, J. Jiang, V. Ng, and X. Wan, Eds. Hong Kong, China: Association for Computational Linguistics, nov 2019, pp. 3982–3992.
- [58] M. Röder, A. Both, and A. Hinneburg, "Exploring the Space of Topic Coherence Measures," in *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, ser. WSDM '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 399–408.
- [59] B. Rosner, R. J. Glynn, and M.-L. T. Lee, "The wilcoxon signed rank test for paired comparisons of clustered data," *Biometrics*, vol. 62, no. 1, pp. 185–192, 07 2005.
- [60] E. Sagdic, A. Bayram, and M. R. Islam, "On the Taxonomy of Developers' Discussion Topics with ChatGPT," in *Proceedings of the 21st International Conference on Mining Software Repositories*, ser. MSR '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 197–201.
- [61] A. J. Scott and M. Knott, "A cluster analysis method for grouping means in the analysis of variance," *Biometrics*, pp. 507–512, 1974.
- [62] J. Shin, C. Tang, T. Mohati, M. Nayeibi, S. Wang, and H. Hemmati, "Prompt engineering or fine-tuning: An empirical assessment of llms for code," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, 2025, pp. 490–502.
- [63] M. S. Siddik and C.-P. Bezemer, "Do code quality and style issues differ across (non-)machine learning notebooks? yes!" in *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2023, pp. 72–83.
- [64] M. L. Siddiq, L. Roney, J. Zhang, and J. C. D. S. Santos, "Quality assessment of chatgpt generated code and their use by developers," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 152–156.
- [65] L. D. Silva, J. Samhi, and F. Khomh, "ChatGPT vs LLaMA: Impact, Reliability, and Challenges in Stack Overflow Discussions," 2024.
- [66] I. R. d. S. Simões and E. Venson, "Evaluating Source Code Quality with Large Language Models: a comparative study," in *Proceedings of the XXIII Brazilian Symposium on Software Quality*, ser. SBQS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 103–113.
- [67] Software Evolution Analytics Lab, "Codechat repository," <https://github.com/Software-Evolution-Analytics-Lab-SEAL/CodeChat.git>, 2025, accessed: 2025-09-01.
- [68] C. Tao, Q. Zhan, X. Hu, and X. Xia, "C4: contrastive cross-language code clone detection," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ser. ICPC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 413–424.
- [69] R. Tufano, A. Mastropaolo, F. Pepe, O. Dabic, M. Di Penta, and G. Bavota, "Unveiling ChatGPT's Usage in Open Source Projects: A Mining-based Study," in *Proceedings of the 21st International Conference on Mining Software Repositories*, ser. MSR '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 571–583.
- [70] L. Twist, J. M. Zhang, M. Harman, D. Syme, J. Noppen, H. Yannakoudakis, and D. Nauck, "A study of llms' preferences for libraries and programming languages," 2025. [Online]. Available: <https://arxiv.org/abs/2503.17181>
- [71] G. van Rossum, B. Warsaw, and A. Coghlan, "Pep 8 – style guide for python code," <https://peps.python.org/pep-0008>, 2025, accessed: 2025-09-01.

- [72] A. Vargha and H. D. Delaney, “The kruskal-wallis test and stochastic homogeneity,” *Journal of Educational and Behavioral Statistics*, vol. 23, no. 2, pp. 170–192, 1998.
- [73] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, “Attention is All you Need,” in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017.
- [74] S. M. Vieira, U. Kaymak, and J. M. C. Sousa, “Cohen’s kappa coefficient as a performance measure for feature selection,” in *International Conference on Fuzzy Systems*, 2010, pp. 1–8.
- [75] Wenting Zhao and Xiang Ren and Jack Hessel and Claire Cardie and Yejin Choi and Yuntian Deng, “Wildchat: 1 million chatgpt interaction logs in the wild,” <https://huggingface.co/datasets/allenai/WildChat>, 2025, accessed: 2025-09-01.
- [76] T. Xiao, C. Treude, H. Hata, and K. Matsumoto, “Devgpt: Studying developer-chatgpt conversations,” in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 227–230.
- [77] Y Combinator, “Hacker news,” <https://news.ycombinator.com/>, 2025, accessed: 2025-09-01.
- [78] M. Yang, Y. Zhou, B. Li, and Y. Tang, “On code reuse from stack-overflow: An exploratory study on jupyter notebook,” *arXiv preprint arXiv:2302.11732*, 2023.
- [79] H. Zhang, S. Wang, H. Li, T.-H. Chen, and A. E. Hassan, “A study of c/c++ code weaknesses on stack overflow,” *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2359–2375, 2022.
- [80] W. Zhao, X. Ren, J. Hessel, C. Cardie, Y. Choi, and Y. Deng, “WildChat: 1M ChatGPT Interaction Logs in the Wild,” *arXiv preprint arXiv:2405.01470*, may 2024.
- [81] L. Zheng, W.-L. Chiang, Y. Sheng, T. Li, S. Zhuang, Z. Wu, Y. Zhuang, Z. Li, Z. Lin, E. P. Xing *et al.*, “Lmsys-chat-1m: A large-scale real-world llm conversation dataset,” *arXiv preprint arXiv:2309.11998*, 2023.
- [82] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li, T. Su, Z. Yang, and J. Tang, “CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X,” in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, ser. KDD ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 5673–5684.
- [83] S. Zhong, Y. Zou, and B. Adams, “Codechat,” <https://huggingface.co/datasets/Suzhen/CodeChat>, 2025, accessed: 2025-09-01.