What Do Developers Ask? A Taxonomy of Code Related Questions

Anonymous ACL submission

Abstract

Progress in Code Question Answering (CodeQA) depends on benchmarks that reflect what developers actually ask. Yet most existing CodeQA datasets are built from docstring templates or GitHub issues, and rarely capture the diversity of real, repository-level developer queries. We address this gap by releasing a taxonomy of real-world code-assistance queries, grounded in 3,052 authentic user interactions from the RepoChat dataset. We construct the taxonomy via a semi-automated pipeline that combines clustering, LLM-based labeling, and an iterative refinement loop driven by inter-model disagreement, and we compare the result against three prior taxonomy-generation methods (TaxoGen, TaxoCom, TaxoAdapt). Our taxonomy achieves a favorable tradeoff between granularity and usability—86 nodes with 93.2% node relevance, 99.4% path validity, and 81.25% sibling coherence—while remaining concise enough for downstream use. We then apply the taxonomy in two ways: (i) auditing existing Code related benchmarks to surface systematically underrepresented query types, and (ii) generating a synthetic CodeQA dataset whose categories are guided by the taxonomy.

1 Introduction

Consider three queries a developer might send to a code assistant working over a specific repository:[1]

- *"Why does the auth module use a custom hashing algorithm instead of bcrypt?"*
- *"Which files do I need to modify to add a new 'Guest' user role across this repo?"*
- *"What environment variables and compiler flags do I need to build this on Linux?"*

[1]Examples illustrative of the intent categories observed in the RepoChat (LMArena (lmarena-ai), 2025) conversations our taxonomy is built on; representative rather than verbatim. See the released dataset for the underlying queries.

None of these queries map cleanly onto existing CodeQA benchmarks. The first asks about *design rationale*—the kind of opinion-flavored "why" question that Stack Overflow's moderation norms actively discourage and that template-derived QA datasets do not produce. The second asks for *repository-wide investigation* grounded in a specific codebase, not a self-contained snippet. The third asks about *repo-specific environment setup*, which prior taxonomies of developer questions, grounded in public Q&A or generated from docstring comments, do not surface. Yet these are exactly the questions developers now ask LLM-based code assistants—and to evaluate and design such assistants, we need a vocabulary that captures them.

Code Question Answering (CodeQA) is increasingly central to developer productivity, code comprehension, and navigation of large codebases (Marcus and Maletic, 2004; LaToza and Myers, 2010). Yet a persistent limitation runs through the field: the queries on which CodeQA systems are trained and evaluated do not closely resemble the queries developers actually ask. Existing benchmarks are largely built by pairing methods with their docstrings (Liu and Wan, 2021), by scraping Stack Overflow snippet-level questions, or by mining GitHub issues—none of which capture the multi-file, context-dependent, often investigative questions that arise when a developer is trying to understand or modify an unfamiliar repository (LaToza and Myers, 2010).

To make progress on this misalignment, the community first needs a shared vocabulary for *what kinds of questions developers actually ask*. Prior taxonomies of developer questions are grounded in Stack Overflow (Allamanis and Sutton, 2013; Beyer et al., 2020) or in template-generated QA pairs (Liu and Wan, 2021), both of which systematically exclude the multi-file, context-grounded, conversational queries that arise with modern LLM-

based code assistants. We discuss this contrast in detail in Section 2.2.

This paper addresses that gap. Our research question is: *how can the diverse range of real-world developer queries for code assistance be conceptualized in a taxonomy, and what does such a taxonomy reveal about existing CodeQA benchmarks?* Our contributions are: (1) **a taxonomy of real-world CodeQA queries** (Figure 6, Appendix H), built from 3,052 authentic developer interactions in the RepoChat dataset (LMArena (lmarena-ai), 2025), organized into 86 nodes across multiple hierarchical levels; (2) **a methodological observation about hierarchical taxonomy refinement:** we identify a *leaf-label bias* that affects human-driven refinement of multi-level taxonomies—once annotators have seen the leaf labels, their judgments at intermediate levels become biased toward reaching the "correct" leaf, masking ambiguity that should drive refinement. We show that multi-LLM traversal, in which each model sees only one level of labels at a time, mitigates this bias and yields a more useful refinement signal; (3) **a semi-automated construction pipeline** (Figure 1) that combines clustering, LLM-based labeling, and the multi-LLM disagreement-driven refinement loop described above; (4) **a comparison against prior taxonomy-generation methods** (Table 2), showing that our taxonomy occupies a useful middle ground between TaxoCom's sparsity (7 nodes) and TaxoAdapt's redundancy (404 nodes) while scoring competitively on node relevance, path validity, and sibling coherence; (5) **an audit of 11 existing CodeQA benchmarks** through the taxonomy (Figure 5, Appendix F), quantifying which developer query types are over- and under-represented; and (6) **an illustrative synthetic dataset** of 426 code questions (Figure 3) generated using the taxonomy's leaf categories, demonstrating one applicability path to data generation.

2 Related Work

2.1 Taxonomy Creation for User Queries

Automated taxonomy generation from text corpora is well-established. TaxoGen (Zhang et al., 2018) introduced unsupervised topic-taxonomy construction using TF-IDF (Salton and Buckley, 1988) and frequency-based clustering, but such methods transfer poorly to query-based taxonomies: high-frequency interrogative tokens (*what*, *how*, *why*) dominate the representations and yield broad, less meaningful categories. TaxoCom (Lee et al., 2022) is a corpus-driven taxonomy completion framework that recursively expands a handcrafted seed taxonomy by clustering corpus terms.

More recently, LLMs have been used to generate taxonomies from user interactions. Shah et al. (2024) analyzed Bing AI chat logs by combining GPT-4-generated candidate taxonomies with iterative expert refinement, validating the result through human–GPT-4 inter-coder agreement; however, their categories are single-level and target general AI queries rather than code-related developer needs. TaxoAdapt (Kargupta et al., 2025) dynamically adapts an LLM-generated taxonomy to a corpus across multiple dimensions, performing iterative hierarchical classification to expand the taxonomy's width and depth according to the corpus's topical distribution.

2.2 Prior Taxonomies of Developer Questions

Several prior works categorize developer questions, though none target conversational assistants operating on specific repositories. Allamanis and Sutton (2013) apply LDA topic modeling to Stack Overflow posts, yielding three flat axes (question concepts, code topics, phrase clusters) organized by technology domain rather than developer intent. Beyer et al. (2020) harmonize five taxonomies into a flat seven-category intent-oriented schema (*API change*, *API usage*, *Conceptual*, *Discrepancy*, *Learning*, *Errors*, *Review*); while close to our work in spirit, the schema lacks sub-categories and remains grounded in Stack Overflow. Liu and Wan (2021) construct CodeQA by extracting template-based questions from code comments, forming a syntactic structure rather than an intent taxonomy.

These works share a common limitation for our setting: they are grounded either in Stack Overflow (Allamanis and Sutton, 2013; Beyer et al., 2020) or in template-generated QA pairs (Liu and Wan, 2021). Both sources systematically exclude question types that arise when developers converse with LLM-based code assistants over a specific repository—design-rationale (*"why is this implemented as a factory pattern here?"*), repository-wide investigation (*"what does this codebase do at a high level?"*), opinion/review (*"can you suggest improvements to this function?"*), code-integration (*"how do I use this library with my Flask app?"*), and repo-specific environment-setup (*"what Java version does this repository need?"*). Stack Overflow moderation discourages opinion-based, overly

Table 1: Survey of 11 existing CodeQA datasets, categorized by data source. None capture authentic, repository-level, conversational queries from developer-assistant interactions. See Appendix G for full descriptions.

Source	Dataset	Primary Question Type
Stack Overflow	ProCQA (Li et al., 2024) COIR (Li et al., 2025)	Bug solving, how-to Code/text retrieval
GitHub issues/commits	CodeRepoQA (Hu et al., 2024) CoReQA (Chen et al., 2025) MegaDiff (Monperrus et al., 2021) CoDocBench (Pai et al., 2025)	Debugging, repo investigation Debugging, investigation Code change categorization Code-doc alignment
Templated / synthetic	CodeQA (Liu and Wan, 2021) SpyderCodeQA (Strich et al., 2024) CodePlan (Bairi et al., 2023)	Investigative, template-based Semantics, dependencies Setup/installation
Curated / MCQs	CodeApex (Fu et al., 2024) RepoChat-Arena (LMArena (lmarena-ai), 2025)	Programming comprehension Repository-level conversation

broad, and codebase-specific questions; docstring-derived templates cannot produce rationale or integration questions.

Our hierarchical, intent-organized taxonomy (86 nodes) is, to our knowledge, the first grounded in this interaction modality. It introduces novel top-level categories—*Code and Project Integration*, *Repository Understanding and Summary*, *Understanding Code Reasoning and Decision Rationale*—with no analog in prior SO-grounded or template-based schemas. While Beyer et al. (2020)'s categories map onto a subset of ours (e.g., their *Review* aligns with our *Code Quality and Security Review*), many of our top-level intents have no counterpart. Our work is thus complementary: prior taxonomies characterize questions asked on public Q&A platforms, while we characterize questions asked of LLM-based assistants about a specific repository.

2.3 Analysis of CodeQA Benchmarks and Datasets for Real Developer Queries

Many CodeQA datasets (see Table 1) are created by scraping sources such as Stack Overflow or GitHub repositories. While valuable, these datasets often lack explicit attention to the diversity of real developer information needs. Existing benchmarks rarely validate whether major query types are sufficiently represented, and most focus on isolated functions or single files rather than repository-level, multi-file, and context-dependent tasks. As a result, complex and scenario-specific developer challenges remain underrepresented (LaToza and Myers, 2010). Our taxonomy provides a structured lens for systematically analyzing and categorizing such queries, enabling assessment of dataset coverage and identification of gaps. This perspective not only improves evaluation of existing benchmarks but also motivates the creation of more diverse and realistic datasets aligned with real developer needs.

3 Methodology

3.1 Dataset selection

We use the RepoChat dataset (LMArena (lmarena-ai), 2025) as our source corpus. We chose RepoChat after surveying 11 prominent CodeQA datasets (Table 1); none capture the authentic, repository-level, conversational queries our taxonomy targets, for the reasons summarized below.

3.1.1 Limitations of Existing CodeQA Datasets

Our survey of 11 prominent datasets (Table 1) reveals three recurring limitations. (1) **Missing conversational context:** GitHub-issue datasets (e.g., CodeRepoQA) include code and issue text but miss the investigative back-and-forth of real developer queries; most GitHub benchmarks also target repository-level code completion rather than question answering. (2) **Superficiality:** Stack Overflow datasets are dominated by snippet-level, bug-fix-heavy questions (e.g., *"How to do X in Python?"*) that rarely require multi-file reasoning or repository-wide understanding. (3) **Template-induced bias:** datasets generated from code summaries and docstrings (e.g., CodeQA) rely on predefined templates that fail to capture the diversity and phrasing of real user queries.

The RepoChat Dataset RepoChat (LMArena (lmarena-ai), 2025) addresses these gaps: users ask code-related questions of an LLM assistant that has access to a user-provided GitHub repository, and the maintainers publicly released roughly

4,000 user-model conversations. We use 3,052 user queries extracted from this corpus.

3.2 Motivation for a Hierarchical Approach

We first tested whether flat unsupervised clustering could group RepoChat queries into intent-aligned categories. Using ModernBERT (Warner et al., 2024) embeddings, we applied K-Means (MacQueen, 1967), DBSCAN (Ester et al., 1996), HDBSCAN (Campello et al., 2013), and BERTopic (Grootendorst, 2022), with GPT-4o generating a representative label per cluster from up to 25 sampled queries. Pairwise agreement across methods was consistently low (Table 4): Adjusted Rand Score (Hubert and Arabie, 1985) and NMI (via scikit-learn (Pedregosa et al., 2011)) were near zero, with K-Means producing 4–7 coarse clusters and BERTopic 55–67 fine-grained ones. This is consistent with prior observations (Bernhard et al., 2023) that mathematically optimal clusters often miss the semantic dimensions a user cares about.

Manual inspection revealed the methods were operating at different granularities: K-Means produced broad themes (e.g., *Code Implementation*, *Code Understanding*) while BERTopic produced specific subtopics (e.g., *API-related questions*, *File I/O*). Two queries—one about an OpenAI API call, one about file handling—would collapse into a single K-Means cluster but separate cleanly under BERTopic. This complementary granularity motivates our pipeline: coarse K-Means clusters become parent nodes, fine BERTopic clusters become their children, yielding a hierarchical taxonomy refined by the iterative loop in Section 4.

4 Taxonomy Generation and Iterative Refinement

Given a set of natural language questions, we aim for a taxonomy with five properties: *hierarchical validity* (meaningful parent-child relationships), *intent alignment* (categorization matches user intent), *sibling coherence* (children consistent with parent), *sibling diversity* (minimal overlap between siblings), and *coverage* (general enough without overspecialization). To achieve this, we developed a semi-automatic methodology based on iterative refinement of candidate taxonomies. The process uses clustering techniques to generate an initial structure of the taxonomy, which is then refined through multiple iterations leveraging the reasoning capability of LLMs to improve coherence and reduce ambiguity.

4.1 Initial Taxonomy Construction

Following the granularity argument in Section 3.2, we construct the initial taxonomy by stitching K-Means and BERTopic outputs: coarse K-Means clusters serve as parent categories, and BERTopic clusters are assigned as children under the most semantically aligned K-Means parent (with intermediate parents introduced where needed). GPT-4 generates labels for each cluster and proposes the parent–child assignments. The full procedure is given in Algorithm 1 (Appendix A).

4.1.1 Related Work on Iterative Taxonomy Refinement

There has been prior work on refining taxonomies in an iterative manner, often leveraging human feedback to resolve ambiguities. For example, Shah et al. (2024) proposed a framework where a taxonomy is constructed and then refined based on annotator disagreement. Specifically, annotators labeled the same set of questions independently, and regions with high disagreement were flagged as ambiguous, suggesting overlap or confusion between taxonomy nodes. These ambiguous regions were then refined to improve clarity and coherence. Our work takes inspiration from this approach, adopting a similar idea of using feedback to identify which parts of the taxonomy require modification and refinement.

4.1.2 Leaf-Label Bias in Human-Driven Refinement

Prior work detects taxonomy ambiguity through inter-annotator disagreement (Shah et al., 2024): multiple annotators independently label the same queries, and regions with high disagreement are treated as candidates for refinement. The underlying intuition is that if annotators consistently disagree, the taxonomy lacks sufficient coherence or diversity in that region. We identify a subtle but important limitation of this approach when applied to deep, multi-level hierarchies. Once annotators are exposed to the full taxonomy—especially the leaf nodes—their judgments at intermediate levels become influenced by knowledge of the final destination. Rather than evaluating each level independently, annotators often select intermediate labels that lead to the leaf node they believe is ultimately "correct." This creates several problems: (1) in hierarchical taxonomies, intermediate nodes

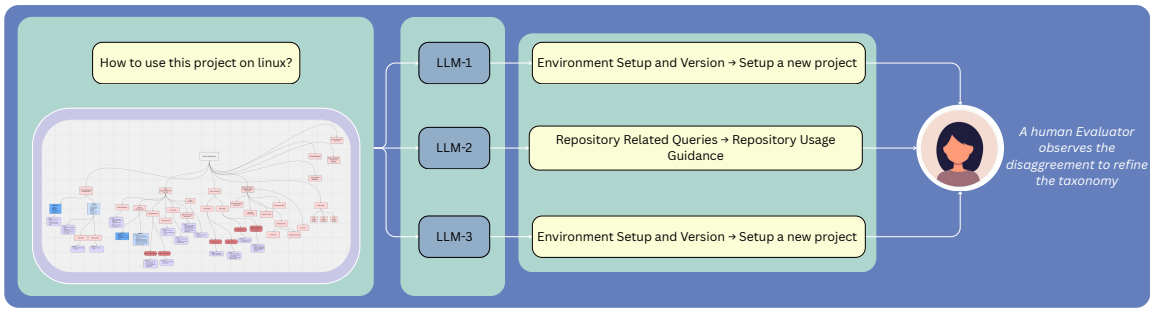

Figure 1: Pipeline for taxonomy generation via clustering and LLM prompting.

are important semantic units whose coherence and diversity should be evaluated independently; (2) after internalizing the leaf structure, annotators find it difficult to disregard downstream labels when making intermediate-level decisions; and (3) as a result, agreement at intermediate levels becomes artificially inflated, obscuring ambiguities that should otherwise motivate taxonomy refinement.

4.1.3 LLMs as Unbiased Traversal Agents

To avoid leaf-label bias, we use multiple Large Language Models as traversal agents. At each level of the taxonomy, each LLM sees only the candidate labels at that level and classifies the query without exposure to deeper levels. This produces level-by-level classifications uncontaminated by leaf knowledge, allowing us to **detect ambiguity** at intermediate levels, **surface overlapping** or weakly differentiated sibling categories, and **obtain finer-grained refinement signals** than leaf-only disagreement provides. We use three LLMs from different model families—Mixtral 8×22B, Qwen2.5-72B-Instruct, and LLaMA-3.3-70B-Instruct—to reduce the risk that disagreement reflects shared architectural bias rather than genuine taxonomy ambiguity. We define a query as *majority-agreed* if at least two of the three models assign it the same label, *partially ambiguous* if exactly two agree, and *no-majority* (or *highly ambiguous*) if all three disagree. The `partially_ambiguous` and `highly_ambiguous` sets drive the refinement loop given in Algorithm 2 (Appendix A).

5 Generation of a Taxonomy for Code Related Queries

We apply the methodology from Section 4 to 3,052 user queries extracted from the RepoChat dataset (LMArena (lmarena-ai), 2025). Algorithm 1 produces an initial taxonomy T1, which we then refine over four iterations using Algorithm 2. At each iteration, we use the size of the *no-majority* set—queries on which all three traversal LLMs disagree on the top-level label—as our refinement signal. A decreasing no-majority count across iterations indicates that the taxonomy is converging toward categories the models can apply consistently; an increasing count indicates that the most recent edits introduced semantic conflicts.

Refinement progression. T1 left 109 queries in the no-majority set. Manual inspection by two authors surfaced three recurring intent patterns absent from the initial K-Means/BERTopic structure: documentation generation, environment setup and usage, and subjective review or opinion requests (e.g., *"Can you suggest improvements to this function?"*). T2 introduced a new top-level branch, *Code Review / Opinion / Feedback*, covering review requests, style opinions, and quality assessments, dropping the no-majority count from 109 to 92.

The remaining no-majority queries concentrated around system-level configuration questions (e.g., *"How do I run this repository on Linux?"*), inconsistently assigned between *Repository-Related Queries* and ad hoc setup leaves. T3 added an explicit *Environment and Setup Issues* branch covering configuration, dependency installation, and runtime/platform issues. After T3, the no-majority count fell to 61, and 1,985 of 2,685 queries at the first level (74%) received unanimous labels across the three traversal LLMs. The full progression is shown in Figure 2.

T4 and T5. We tested two further iterations consisting primarily of label renaming intended to reduce sibling overlap. Disagreement increased in both cases, as renamed labels introduced semantic conflicts with adjacent categories: for example, a renamed leaf could now plausibly fit under two parents, splitting model assignments. We treat this as the convergence signal that Algorithm 2's stop-

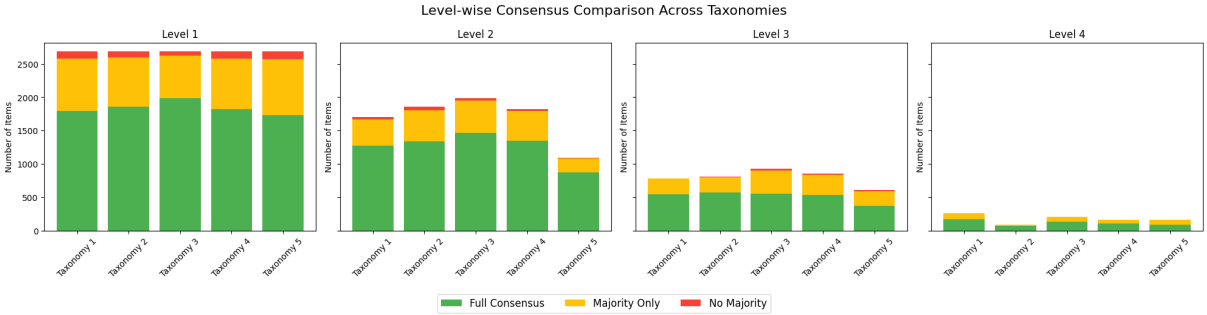

Figure 2: Agreement statistics across taxonomy iterations T1–T5, grouped by inter-LLM agreement level (*full consensus*, *majority only*, *no majority*) at each level of the taxonomy. The no-majority count drops from 109 (T1) to 92 (T2) to 61 (T3), then rises under T4–T5 as label renaming introduced semantic conflicts; we retain T3. Agreement is computed across three traversal LLMs (Mixtral 8×22B, Qwen2.5-72B-Instruct, LLaMA-3.3-70B-Instruct).

ping criterion calls for, and retain T3 as our final taxonomy (Figure 6, Appendix H).

5.1 External Validation on an Internal COBOL Dataset

We evaluated the refined taxonomy on a private dataset of 321 COBOL-related queries from an internal CodeQA system, to test generalizability beyond RepoChat. Of the 321 queries, 213 (66%) received unanimous labels across the three traversal LLMs, and only 15 (5%) received no agreement at all. This supports the taxonomy's transferability to at least one other language and domain.

One pattern of disagreement was informative. Questions like "What is the required Java version for this repository?" were variously labeled as *Repository-Related Queries* or *Environment and Setup Issues* by different models. Originally categorized under the former, this confusion highlighted a genuine ambiguity in T3 between repository metadata queries and environment-configuration queries. We resolved it by renaming *Environment and Setup Issues* to *Environment Setup and Code Version-Related Queries*, making code-version inquiries explicitly part of that branch. This exemplifies the broader pattern of the refinement loop: external disagreement signal directly drove a concrete taxonomy improvement.

6 Comparison with Other Taxonomy Creation Methods

We compare our taxonomy against three prior taxonomy-generation methods: TaxoGen (Zhang et al., 2018), TaxoCom (Lee et al., 2022), and TaxoAdapt (Kargupta et al., 2025).

6.1 Evaluation Metric

We use an automatic evaluation suite with two LLM judges—Gemini 2.5 Pro (DeepMind, 2025) and GPT-4o (OpenAI, 2024)—to assess the quality of generated taxonomies along both node-level and taxonomy-level dimensions, following the evaluation framework of (Kargupta et al., 2025). For each metric, we average the per-node scores produced by the two judges before aggregating to the taxonomy level. We report the averaged numbers in Table 2.

Note on size disparity across baselines. The compared methods produce taxonomies of substantially different sizes: TaxoCom (Lee et al., 2022) 7 nodes, ours 86, TaxoAdapt (Kargupta et al., 2025) 404. We ran each method at its default configuration without tuning, so these counts reflect each method's intended operating regime, not our experimental setup. The disparity nonetheless affects raw metrics: small taxonomies trivially achieve high path validity but offer little coverage, while large ones inflate sibling-coherence through redundancy. We position our contribution as occupying a useful middle ground rather than winning any single metric in isolation; we discuss this size–utility tradeoff in Section 7.

Path Validity For each parent-child pair, the judge assigns 1 if the child is a logically more specific sub-category, 0 otherwise. This is a binary metric of hierarchical consistency. Full rubric: Appendix C.

Node Relevance and Intent Alignment For each node, the judge assigns 2 (highly relevant, well-framed), 1 (vaguely related or poorly framed), or 0 (irrelevant). The judge also produces a struc-

Table 2: Comparison of Taxonomies across Evaluation Metrics (Best in **bold**, 2nd best underlined)

Taxonomy	Path Validity (%)	Node Relevance (%)	Sibling Coherence (%)	Average Score (%)	Nodes	Leaves
TaxoGen	47.2	23.4	16.76	29.12	341	256
TaxoAdapt	86.1	84.3	**89.11**	86.50	404	311
TaxoCom	**100.0**	57.7	66.66	74.79	7	4
Our Taxonomy	99.4	**93.2**	81.25	**91.28**	86	62

Scores are the average of Gemini 2.5 Pro (DeepMind, 2025) and GPT-4o (OpenAI, 2024).

tured critique covering label sanity and keyword relevance. Full rubric: Appendix C.

Sibling Coherence For each parent node p, the judge scores its sibling set S_p: 2 (coherent: siblings distinct, related, same granularity), 1 (partial coherence), or 0 (incoherent or mixed granularity). Full rubric: Appendix C. To report the final aggregated metrics in Table 2 and Table 3, we normalized the raw scores into percentages by dividing the total score received by the maximum possible score for each respective metric.

6.2 Human Validation of LLM Judges

To validate our use of LLM judges, Seven annotators with software-engineering backgrounds independently scored a stratified sample of 104 items across all four taxonomies: 50 parent–child pairs (path validity), 34 nodes (node relevance), and 20 sibling sets (sibling coherence). Annotators were blinded to source taxonomy and to the LLM judges' scores. Fleiss' κ was 0.19, 0.11, and 0.13 respectively, reflecting the inherent subjectivity of taxonomy quality judgments.

Results. Mean human scores (as a percentage of maximum possible) are shown alongside LLM-judged scores in Table 3. On path validity, the human ranking matches the LLM ranking (Ours > TaxoAdapt > TaxoGen > TaxoCom). On node relevance, our taxonomy and TaxoAdapt score comparably under human judgment (76.2% vs. 76.4%), both well above TaxoCom and TaxoGen. On sibling coherence, human scores for our taxonomy (81.2%) align closely with the LLM-judged score (81.25%).

Discussion. Human scores are systematically lower than LLM scores for TaxoCom and TaxoGen. We attribute this to a known limitation of LLM-as-judge evaluation: LLM judges can validate syntactic hierarchy in isolation (e.g., *use* → *answers* passes a structural check) even when labels are semantically opaque to a human reader. As illustrated in Figure 4, TaxoCom's keyword-cluster nodes (e.g., *use*, *incorporate*, *answers*) and TaxoGen's token-cluster nodes (e.g., `cython_tensorflow_opencv_numpy_triton`) are difficult for annotators to interpret as developer-question categories, yielding low human scores despite passing LLM structural checks. Our taxonomy and TaxoAdapt, whose nodes are intent-labeled in natural language, score comparably under both judge types—suggesting that human and LLM judges converge when node labels are semantically transparent. This supports the use of LLM judges for comparing intent-based taxonomies while cautioning that LLM judges may overestimate the quality of keyword- or token-cluster taxonomies. Relative comparisons in Table 2 remain valid since all taxonomies are scored by the same judges.

Table 3: Mean human scores (%) per taxonomy and metric. Human scores are averaged across 7 annotators

Taxonomy	Path Validity	Node Relevance	Sibling Coherence
TaxoGen	29.9	39.3	41.7
TaxoAdapt	62.9	**76.4**	73.8
TaxoCom	17.9	38.1	41.7
Ours	**82.7**	76.2	**81.2**

7 Analysis of Results

7.1 Experimental Setup

TaxoGen(Zhang et al., 2018) and TaxoCom(Lee et al., 2022) do not require any explicit parameter related to user intent when generating their taxonomies. In contrast, TaxoAdapt(Kargupta et al., 2025) is an LLM-based method, which requires a user intent parameter as input. All methods were evaluated after running their code on 3,052 questions from the RepoChat dataset (LMArena (lmarena-ai), 2025). In addition, these meth-

ods come with specific hyperparameter requirements. For instance, TaxoGen requires parameters such as the number of children per parent, while TaxoAdapt requires settings like `max_depth` and `max_density`. These choices influence the granularity and overall usability of the resulting taxonomies.

7.2 Results and Discussion

Table 2 and Figure 4 reveal a fundamental size–utility tradeoff. TaxoCom's 7 nodes (4 leaves) trivially achieve 100% path validity but cannot disambiguate real developer query types: its two top-level branches (*use*, *incorporate*) and single-word leaves (*answers*, *source*, *list*, *model*) are too coarse for downstream use. TaxoGen's 341 nodes are largely uninterpretable, with self-referential loops (e.g., *unusual_way* → *unusual_way* → *unusual_way*) and token-cluster labels (e.g., `cython_tensorflow_opencv_numpy_triton`), reflecting frequency-based clustering on noisy query text. TaxoAdapt's 404 nodes (311 leaves) from only 3,052 queries exhibit a redundancy problem: similar intents duplicate across branches (e.g., *debugging_runtime_errors* and *troubleshooting_runtime_errors* as separate nodes), which artificially inflates its sibling-coherence score by keeping each branch internally consistent while forcing arbitrary labeling choices and requiring manual deduplication for benchmark use. Its coherence score should therefore be read alongside its node count.

Our taxonomy achieves 99.4% path validity, 93.2% node relevance, and 81.25% sibling coherence across 86 nodes (62 leaves)—large enough to capture 10 distinct top-level intents without redundancy, compact enough that every node is human-interpretable and usable without disambiguation. The slight coherence gap relative to TaxoAdapt is a deliberate consequence of broader, non-redundant categories per parent.

8 Application of Taxonomy Produced

8.1 Analysis of Code Related Benchmarks and Datasets

As discussed in Section 3.1.1, our survey of 11 popular CodeQA datasets (Table 1) revealed a recurring mismatch between what developers actually ask and what these datasets represent. GitHub-based datasets focus on issues and commits rather than developer assistance; Stack Overflow-based ones emphasize low-level code questions and lack repository-wide or design-level context; and synthetic datasets are template-driven, failing to capture genuine developer intent.

Question Diversity Analysis Using our Taxonomy We analyzed SWA Bench (Vergopoulos, 2025), SWE Bench (Jimenez et al., 2023) (test and dev splits), and SpyderCodeQA (Strich et al., 2024) by classifying each question against our taxonomy and plotting the resulting distribution (Figure 5, Appendix F).

Underrepresented Categories. Two categories are particularly underrepresented in current benchmarks despite being common in practice: (1) **Intent Behind Code**—design-rationale questions like *"Why is this done this way and not another?"* are rare in templated datasets; and (2) **Code Integration**—questions like *"How do I use this API with my framework?"* don't surface from snippet-level Stack Overflow scraping.

8.2 Illustrative Use Case: Taxonomy-Guided Synthetic Data Generation

As a second illustrative use case, we use leaf-node categories from our taxonomy as structured prompt components for synthetic question generation, producing 426 questions across 19 leaf categories using mistral-small-3.1-24b-instruct over four open-source Python repositories. We emphasize this is a feasibility demonstration rather than a controlled evaluation. Representative examples are shown in Figure 3; full setup, prompt details, and scope limitations are in Appendix D.

9 Conclusion and Future Work

We address the gap between existing CodeQA datasets and authentic developer queries by introducing a semi-automated methodology for constructing and iteratively refining query taxonomies. Applying it to RepoChat (LMArena (lmarena-ai), 2025), we produced a taxonomy that enables rigorous benchmark auditing and taxonomy-guided synthetic data generation, achieving a strong balance between sparsity, redundancy, and sibling coherence. Future work will focus on cross-platform validation and human inter-annotator studies to validate our LLM-as-judge metrics and refinement decisions. A natural methodological next step is developing an automated refinement agent to reduce human review effort, alongside richer evaluation-grounded synthetic data generation strategies.

10 Limitations

Researcher bias in manual refinement. The manual refinement stages (Step 9 of Algorithm 1, Step 17 of Algorithm 2) introduce potential researcher bias. We mitigate this by grounding all manual changes in observed LLM disagreement: refinements were targeted only at the `partially_ambiguous` and `highly_ambiguous` sets surfaced by Algorithm 2. Two authors with software-engineering backgrounds performed the review across all five iterations—one drafting, the other reviewing, with disagreements resolved by discussion. A formal Cohen's κ study of refinement decisions is left to future work.

Dependence on Large Language Models. We rely on LLMs for label generation (GPT-4) and ambiguity detection (Mixtral 8×22B, Qwen2.5-72B-Instruct, LLaMA-3.3-70B-Instruct). To reduce shared-bias risk in the latter, we selected models from three different families; replication with additional families remains future work.

Generalization beyond RepoChat. Our taxonomy is grounded in RepoChat and was not evaluated on (i) non-English queries, (ii) IDE-integrated assistant logs (e.g., Copilot Chat, Cursor), (iii) low-code, notebook, or DSL queries, or (iv) enterprise-internal codebases. We did evaluate on a private dataset of 321 COBOL queries (Section 5.1), where 213/321 (66%) received unanimous labels across the three traversal LLMs, supporting transferability to at least one other language and domain. Broader cross-platform validation remains future work.

References

Miltiadis Allamanis and Charles Sutton. 2013. Why, when, and what: analyzing stack overflow questions by topic, type, and code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 53–56. IEEE.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. 2023. Codeplan: Repository-level coding using llms and planning. *Preprint*, arXiv:2309.12499.

Jana Bernhard, Martin Teuffenbach, and Hajo G. Boomgaarden. 2023. Topic model validation methods and their impact on model selection and evaluation. *Computational Communication Research*, 5(1):1.

Stefanie Beyer, Christian Macho, Massimiliano Di Penta, and Martin Pinzger. 2020. What kind of questions do developers ask on Stack Overflow? A comparison of automated approaches to classify posts into question categories. *Empirical Software Engineering*, 25(3):2258–2301.

Ricardo J. G. B. Campello, Davoud Moulavi, and Joerg Sander. 2013. Density-based clustering based on hierarchical density estimates. In *Advances in Knowledge Discovery and Data Mining*, pages 160–172, Berlin, Heidelberg. Springer Berlin Heidelberg.

Jialiang Chen, Kaifa Zhao, Jie Liu, Chao Peng, Jierui Liu, Hang Zhu, Pengfei Gao, Ping Yang, and Shuiguang Deng. 2025. Coreqa: Uncovering potentials of language models in code repository question answering. *Preprint*, arXiv:2501.03447.

Google DeepMind. 2025. Gemini 2.5 pro. `https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-5-pro`. Large multimodal reasoning model.

Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press.

Lingyue Fu, Huacan Chai, Shuang Luo, Kounianhua Du, Weiming Zhang, Longteng Fan, Jiayi Lei, Renting Rui, Jianghao Lin, Yuchen Fang, Yifan Liu, Jingkuan Wang, Siyuan Qi, Kangning Zhang, Weinan Zhang, and Yong Yu. 2024. Codeapex: A bilingual programming evaluation benchmark for large language models. *Preprint*, arXiv:2309.01940.

Maarten Grootendorst. 2022. Bertopic: Neural topic modeling with a class-based tf-idf procedure. *Preprint*, arXiv:2203.05794.

Ruida Hu, Chao Peng, Jingyi Ren, Bo Jiang, Xiangxin Meng, Qinyun Wu, Pengfei Gao, Xinchen Wang, and Cuiyun Gao. 2024. Coderepoqa: A large-scale benchmark for software engineering question answering. *Preprint*, arXiv:2412.14764.

L. Hubert and P. Arabie. 1985. Comparing partitions. *Journal of Classification*, 2(1):193–218.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? `https://huggingface.co/datasets/princeton-nlp/SWE-bench`. Accessed: 2025-09-10.

Priyanka Kargupta, Nan Zhang, Yunyi Zhang, Rui Zhang, Prasenjit Mitra, and Jiawei Han. 2025. Taxoadapt: Aligning llm-based multidimensional taxonomy construction to evolving research corpora. *Preprint*, arXiv:2506.10737.

Thomas D. LaToza and Brad A. Myers. 2010. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, New York, NY, USA. Association for Computing Machinery.

Dongha Lee, Jiaming Shen, SeongKu Kang, Susik Yoon, Jiawei Han, and Hwanjo Yu. 2022. Taxocom: Topic taxonomy completion with hierarchical discovery of novel topic clusters. *CoRR*, abs/2201.06771.

Xiangyang Li, Kuicai Dong, Yi Quan Lee, Wei Xia, Hao Zhang, Xinyi Dai, Yasheng Wang, and Ruiming Tang. 2025. Coir: A comprehensive benchmark for code information retrieval models. *Preprint*, arXiv:2407.02883.

Zehan Li, Jianfei Zhang, Chuantao Yin, Yuanxin Ouyang, and Wenge Rong. 2024. Procqa: A large-scale community-based programming question answering dataset for code search. *Preprint*, arXiv:2403.16702.

Chenxiao Liu and Xiaojun Wan. 2021. Codeqa: A question answering dataset for source code comprehension. *Preprint*, arXiv:2109.08365.

LMArena (lmarena-ai). 2025. Repochat arena preference 4k dataset. https://huggingface.co/datasets/lmarena-ai/repochat-arena-preference-4k. Leaderboard vote data on RepoChat collected from 2024-11-30 to 2025-02-03; licensed under CC-BY-4.0.

James MacQueen. 1967. Some methods for classification and analysis of multivariate observations. *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, 1:281–297.

Andrian Marcus and Jonathan I Maletic. 2004. Information retrieval based approach for locating concept definitions in source code. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC)*, pages 251–259. IEEE.

Mistral AI. 2025. Mistral-small-3.1-24b-instruct-2503. https://huggingface.co/mistralai/Mistral-Small-3.1-24B-Instruct-2503. Instruction-tuned 24B parameter model with 128K context length.

Martin Monperrus, Matias Martinez, He Ye, Fernanda Madeiral, Thomas Durieux, and Zhongxing Yu. 2021. Megadiff: A dataset of 600k java source code changes categorized by diff size. *Preprint*, arXiv:2108.04631.

OpenAI. 2024. Gpt-4o: Omni model. https://openai.com/index/hello-gpt-4o/. Multimodal model (text, audio, image, video).

Kunal Pai, Premkumar Devanbu, and Toufique Ahmed. 2025. Codocbench: A dataset for code-documentation alignment in software maintenance. *Preprint*, arXiv:2502.00519.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine learning in Python. In *Journal of Machine Learning Research*, volume 12, pages 2825–2830.

Gerard Salton and Christopher Buckley. 1988. Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5):513–523.

Chirag Shah, Ryen W. White, Reid Andersen, Georg Buscher, Scott Counts, Sarkar Snigdha Sarathi Das, Ali Montazer, Sathish Manivannan, Jennifer Neville, Xiaochuan Ni, Nagu Rangan, Tara Safavi, Siddharth Suri, Mengting Wan, Leijie Wang, and Longqi Yang. 2024. Using large language models to generate, validate, and apply user intent taxonomies. *Preprint*, arXiv:2309.13063.

Jan Strich, Florian Schneider, Irina Nikishina, and Chris Biemann. 2024. On improving repository-level code QA for large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*, pages 209–244, Bangkok, Thailand. Association for Computational Linguistics.

K. Vergopoulos. 2025. Swa-bench: A benchmark for repository-level code generation. https://huggingface.co/datasets/LogicStar/SWA-Bench. Accessed: 2025-09-10.

Benjamin Warner, Antoine Chaffin, Benjamin Clavié, Orion Weller, Oskar Hallström, Said Taghadouini, Alexis Gallagher, Raja Biswas, Faisal Ladhak, Tom Aarsen, Nathan Cooper, Griffin Adams, Jeremy Howard, and Iacopo Poli. 2024. Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference. *Preprint*, arXiv:2412.13663.

Chao Zhang, Fangbo Tao, Xiusi Chen, Jiaming Shen, Meng Jiang, Brian Sadler, Michelle Vanni, and Jiawei Han. 2018. Taxogen: Unsupervised topic taxonomy construction by adaptive term embedding and clustering. *Preprint*, arXiv:1812.09551.

A Algorithms

Algorithm 1 details the initial taxonomy construction described in Section 4.1. Algorithm 2 details the iterative refinement loop described in Section 4.1.3.

Algorithm 1 Taxonomy Generation via Cluster Stitching

Require: A set of user questions Q
Ensure: A hierarchical taxonomy T

```
Embed all questions Q using ModernBERT to obtain vector representations.
Cluster embeddings using K-Means with k = 10 (selected by qualitative coherence inspection over k ∈ [5, 10]) to obtain coarse clusters C_kmeans.
Apply BERTopic to Q to obtain fine-grained topic clusters C_bertopic.
for all clusters c ∈ C_kmeans ∪ C_bertopic do
    Sample up to 25 representative questions from c.
    Use GPT-4 to generate a semantic label for the sampled questions.
end for
Use GPT-4 to group BERTopic labels under their most semantically aligned K-Means label, introducing intermediate parent nodes where appropriate.
Connect grouped BERTopic labels (or their intermediate parents) as children of the corresponding K-Means label.
Manually review and refine the structure to adjust misclassifications and reassign children as needed.
return Taxonomy T
```

Algorithm 2 Iterative Refinement of Taxonomy Using Multiple LLMs

Require: Set of texts $T = \{t_1, t_2, \ldots, t_n\}$
Require: Set of LLMs $M = \{m_1, m_2, \ldots, m_k\}$
Require: Initial taxonomy $\mathcal{C}$

```
repeat
    for each text t ∈ T do
        for each model m ∈ M do
            Use m to classify t into one class at each level of taxonomy C without introducing bias in the prompt.
        end for
    end for
    Collect classification outputs for each text from all models.
    Initialize sets: partially_ambiguous ← ∅, highly_ambiguous ← ∅.
    for each text t ∈ T do
        if models partially agree (exactly two of three) then
            Add t to partially_ambiguous.
        else if all models disagree then
            Add t to highly_ambiguous.
        end if
    end for
    Use the ambiguous sets to identify and refine unclear or overlapping labels in C.
until the size of partially_ambiguous ∪ highly_ambiguous saturates across iterations.
```

B Cluster Correlation Across Methods

Table 4 reports pairwise agreement between clustering methods applied to ModernBERT embeddings of the RepoChat queries. ARS denotes Adjusted Rand Score; NMI denotes Normalized Mutual Information. The consistently low values across method pairs motivate the hierarchical approach discussed in Section 3.2.

C LLM Judge Rubrics

Each rubric below was used as the prompt instruction given to the two LLM judges (Gemini 2.5 Pro (DeepMind, 2025) and GPT-4o (OpenAI, 2024)) when scoring the four compared taxonomies. The condensed forms appear in Section 6.1.

C.1 Path Validity

For each parent–child pair in a taxonomy, the judge was asked to determine whether the child node

Table 4: Pairwise correlation between clustering and topic-modeling methods on ModernBERT embeddings.

Method 1 (# clusters)	Method 2 (# clusters)	ARS	NMI
K-Means (4)	DBSCAN (14)	0.14	0.33
BERTopic (28)	LLM (7)	0.029	0.156
BERTopic (67)	K-Means (4)	0.015	0.15
BERTopic (66)	K-Means (4)	0.03	0.29
K-Means (7)	LLM (7)	0.137	0.19
BERTopic (56)	LLM (72)	0.017	0.20
BERTopic (55), noise removed	LLM (72)	0.05	0.26
BERTopic (55), noise removed	LLM (44)	0.05	0.25

represents a logically more specific sub-category of the parent. The judge assigned:

- **1** if the child was clearly a more specific type, component, or sub-topic of the parent;
- **0** if the relationship was illogical, inconsistent, or non-hierarchical.

This yielded a binary node-level metric of hierarchical consistency, aggregated over all parent–child pairs in the taxonomy.

C.2 Node Relevance and Intent Alignment

The judge was instructed to act as an impartial taxonomist and developer-experience expert. For each node, it determined whether the node represents an appropriate and well-framed category for classifying developer queries, assigning:

- **2** if the node is highly relevant and well-framed as a developer-question category;
- **1** if the node is vaguely related but ambiguous, poorly framed, or not a valid category;
- **0** if the node is irrelevant or nonsensical with respect to developer intent.

The judge also produced a structured critique covering *Node Name Sanity* (whether the label is intuitive, logical, and representative of a real developer query type) and *Keyword Relevance* (whether keywords under the node logically support the intended meaning).

C.3 Sibling Coherence

For every parent node p, the judge was given the sibling set $S_p = \{c_1, c_2, \ldots\}$ and asked to evaluate whether the siblings form a coherent, same-level set. Scoring:

- **2 (Coherent)**: siblings are distinct, related, and at the same level of specificity (e.g., *Car, Boat, Plane*).
- **1 (Partially coherent)**: the set shows some thematic relation but has ambiguity, mixed granularity, or coverage gaps.
- **0 (Incoherent)**: siblings do not form a logical group or mix abstraction levels (e.g., *Vehicle, Car, SportsCar*).

The judge provided a brief justification covering two criteria: *Coherence* (thematic belonging) and *Consistent Granularity* (level of abstraction).

D Taxonomy-Guided Synthetic Data Generation

This appendix expands the synthetic-data illustrative use case from Section 8.2. We emphasize that this study is intended to demonstrate feasibility, not to claim that taxonomy-guided generation outperforms taxonomy-free alternatives; a controlled comparison is left to future work.

Setup. We used `mistral-small-3.1-24b-instruct` (Mistral AI, 2025) as the question generator. The code corpus was assembled from four widely used open-source Python repositories—`Flask`, `pandas`, `requests`, and `NumPy`—to ground generated questions in realistic code context. Each leaf-node category from our taxonomy was supplied as a `type_of_question` parameter in the prompt. We experimented with two prompt variants: providing only the leaf-node label, and providing the full root-to-leaf path (e.g., `Code Quality and Security Review > Code Security Analysis > Vulnerability Detection`). The two variants produced qualitatively similar outputs; we adopted the full-path variant for the final generation.

Output. The resulting dataset contains 426 synthetic questions spanning 19 leaf categories. Representative examples are shown in Figure 3. We release the dataset alongside this paper to support

future work on taxonomy-grounded synthetic data generation for CodeQA.

Scope and limitations. This study has three deliberate limitations. First, we do not evaluate the quality, realism, or diversity of the generated questions against a baseline, and we make no claim that taxonomy-guided generation improves over taxonomy-free generation. Second, the generator is a single mid-sized model, and outputs may differ with stronger or differently aligned models. Third, the corpus is restricted to four Python repositories. A systematic evaluation along these axes is an important direction for future work.

E Illustrative Taxonomy Snippets from Baseline Methods

Figure 4 shows representative portions of the taxonomies generated by TaxoGen, TaxoAdapt, and TaxoCom on the same 3,052 RepoChat queries. The snippets illustrate the qualitative differences in node naming and structural organization discussed in Section 7.2: TaxoCom's coarse single-word leaves, TaxoGen's token-cluster and self-referential artifacts, and TaxoAdapt's intent-labeled but redundant branches.

F Benchmark Diversity Analysis

Figure 5 (Appendix F) shows the distribution of question types across SWA Bench, SWE Bench (test and dev), and SpyderCodeQA, classified using our taxonomy's leaf categories. The plot confirms that *Intent Behind Code* and *Code Integration* categories are systematically underrepresented across all four benchmarks.

G Full CodeQA Dataset Survey

Table 5 provides the full version of the dataset survey summarized in Table 1, with detailed question types and generation methods for each surveyed benchmark.

H Final Taxonomy

Figure 6 presents the complete T3 taxonomy with all 86 nodes across multiple hierarchical levels, organized under 10 top-level intent categories.

I Use of AI Assistants for Writing and Coding

In accordance with the ACL policies on the use of AI assistants, we acknowledge the use of Large Language Models (such as ChatGPT and Gemini) and AI-assisted coding tools during the preparation of this manuscript. Specifically, these tools were employed to assist with writing routine data-preprocessing scripts, formatting LaTeX elements, and proofreading the manuscript to improve grammatical flow and clarity. All AI-generated text and code were rigorously reviewed, edited, and validated by the authors. The human authors maintain full responsibility for the content, accuracy, originality, and scientific integrity of this work.

Table 5: Survey of 11 existing CodeQA datasets with question types and generation methods.

Dataset	**Question Types**	**Question Generation Method**
CodeQA (Liu and Wan, 2021)	Investigative, repository-level	Template-based questions
COIR (Li et al., 2025)	Code→Text, Text→Code, hybrid retrieval	Aggregated from multiple datasets (incl. StackOverflow)
CodeApex (Fu et al., 2024)	Programming comprehension, code generation, code correction	Comprehension from university-level MCQs; generation from internal Online Judge exercises (questions, samples, templates, graded test cases); correction from real student submissions
CodeRepoQA (Hu et al., 2024)	Debugging, repo-level investigation	GitHub issues
CoReQA (Chen et al., 2025)	Debugging, investigation	GitHub issues
SpyderCodeQA (Strich et al., 2024)	Semantics comprehension, dependency analysis, meta-info retrieval	Mix of manual expert annotation (source code and meta-information) and automated generation via AST analysis and OpenAI API with expert verification (dependencies)
CoDocBench (Pai et al., 2025)	Code–documentation alignment, evolution	Derived from code versioning and documentation
MegaDiff (Monperrus et al., 2021)	Categorized code changes (diff-based)	Git commits
CodePlan (Bairi et al., 2023)	Setup/installation investigation	Manual prompts simulating installation problems
ProCQA (Li et al., 2024)	Bug solving, how-to, why-type	StackOverflow Q&A pairs (filtered for quality)
RepoChat-Arena (LMArena (lmarena-ai), 2025)	Repository-grounded conversational queries	Chat between user and SWE-investigative agent

Taxonomy Category	Example Question
Code Understanding and Explanation > Understanding Code Logic, Data Flow, and Relationships and How does the code/something work? > General Code Logic Explanation	Can you explain the general logic of how the HTTPDigestAuth class in the requests.auth module handles HTTP Digest Authentication? Specifically, how does it manage the state across different threads and what steps does it take to build the Authorization header for a request?
Code Understanding and Explanation > Understanding Code Logic, Data Flow, and Relationships and How does the code/something work? > Code Dependencies	Can you explain the dependencies and the flow of data in the HTTPDigestAuth class, particularly how the build_digest_header method interacts with the handle_401 method? How does the state management with threading local storage work in this context?
Code Understanding and Explanation > Getting to know Purpose and Characteristics of Code Elements > Understanding Performance Characteristics of Code	In the RequestsCookieJar class, the methods __getitem__ and get use the _find_no_duplicates method to retrieve cookie values. This method is described as O(n) due to iterating over all cookies. How does this affect the performance of operations like cookie_jar['cookie_name'] when the cookie jar contains a large number of cookies? Are there any optimizations that could be considered to improve the performance of these operations?
Code Understanding and Explanation > Understanding Code Reasoning and Decision Rationale > Alternative Approaches and Trade-offs	In the Blueprint class, the send_static_file method is used to serve static files from the static_folder. What are some alternative approaches to serving static files, and what are the trade-offs associated with each approach? For example, should we consider using a dedicated static file server like Nginx, or is it better to handle static files within the application itself? How do these choices impact performance, scalability, and maintainability?
Code Location and Bug Identification > Code Finding > Code Finding Based on Name	I'm trying to locate the specific section of the code where the C API borrow-reference linter is executed. Could you help me identify the function and the exact lines of code responsible for running this linter?
Code Understanding and Explanation > Understanding Code Logic, Data Flow, and Relationships and How does the code/something work? > Impact of Code Changes	What would be the impact on the MethodView class if the dispatch_request method's logic were modified to always fallback to the get method for any unimplemented HTTP method, instead of raising an assertion error? Specifically, how would this change affect the behavior of a subclass that does not implement all HTTP methods?
Code Understanding and Explanation > Understanding Code Reasoning and Decision Rationale > Design Decisions and Logic Justifications	Why does the HTTPDigestAuth class use threading to manage state? Specifically, what design decisions led to the use of threading.local for storing state variables like last_nonce, nonce_count, and chal? How does this approach help in handling concurrent HTTP requests?
Code Quality and Security Review > Code Review Requests	In the requests.auth.HTTPDigestAuth class, state is stored using threading.local(). Is this approach secure and thread-safe, and are there better practices or improvements we should consider?
Code Quality and Security Review > Code Security Analysis > Vulnerability Detection	In the send_static_file method, how can an attacker exploit the file handling to access sensitive files outside the intended static_folder?
Code Quality and Security Review > Opinion on Code Style or Design	Considering the Session class in the requests library, how would you evaluate the design choice of using a mixin (SessionRedirectMixin) to handle redirection logic? Do you think this design decision enhances code maintainability and readability, or would it be better to integrate this logic directly into the Session class?

Figure 3: Representative examples of synthetic questions generated using our taxonomy. One example per leaf-node category, drawn from the 426 generated questions across 19 categories.

```
└Root Topics
  ├──use
  │  │  (review, aspects, examine, identify, offer, projects, answers, detailed, structure, comments)
  │  │
  │  ├────── answers
  │  │    (answers, know, points, algebra, learning, bullet, offer, stepbystep, understanding, help)
  │  │
  │  └────── source
  │       (confirm, source, files, based, given, directly, complete, requirements, compatible, executed)
  │
  └incorporate
     │  (chat, server, modify, multiple, include, displayed, incorporate, text, custom, data)
     │
     ├────── list
     │     (number, line, message, messages, item, result, displayed, string, view, value)
     │
     └────── model
        (model, openai, employ, docker, support, local, custom, build, utilize, implement)
```

```
├── Code_Related_Query
│   └── best_practices_inquiry
│       ├── code_efficiency_optimization
│       │   ├──
performance_bottleneck_identification
│       │   ├── algorithm_optimization
│       │   │   ├── performance_analysis
│       │   │   └── tradeoff_evaluation
│       │   ├── resource_management
│       │   ├── code_refactoring
│       │   │   ├── simplify_structure
│       │   │   ├── remove_duplication
│       │   │   └── readability_vs_efficiency
│       │   └── profiling_and_benchmarking
│       │       ├── profiling_tool_selection
│       │       ├── resource_usage_analysis
│       │       ├── optimization_evaluation
│       │       └── benchmarking_protocols
│       ├── security_implementation_guidelines
│       │   ├── secure_coding_practices
│       │   │   ├── vulnerability_prevention
│       │   │   └── compliance_adherence
│       │   ├── authentication_mechanisms
│       │   ├── data_encryption
│       │   ├── access_control
│       │   │   ├── rbac_implementation
│       │   │   ├── abac_best_practices
│       │   │   ├── access_control_analysis
│       │   │   ├── dac_debugging
│       │   │   └── mac_comparison
│       │   └── vulnerability_mitigation
│       ├── patch_management
│       ├── security_testing
│       ├── risk_assessment
│       └── regulatory_compliance
├── scalability_strategies
│   ├── load_balancing
│   │   ├── implementation_steps
│   │   ├── algorithm_selection
│   │   ├── troubleshooting
│   │   └── deployment_best_practices
│   ├── database_scalability
│   │   ├── indexing_optimization
│   │   └── data_partitioning
│   ├── caching_design
│   │   ├── eviction_strategies
│   │   ├── distributed_cache_coherence
│   │   ├── hit_rate_optimization
│   │   ├── invalidation_policies
│   │   └── query_pattern_analysis
│   ├── horizontal_vs_vertical_scaling_trade-offs
│   │   ├── cost_efficiency_analysis
│   │   ├── performance_tradeoff_evaluation
│   │   ├── system_complexity_assessment
│   │   ├── resource_utilization_trade-offs
│   │   └── implementation_feasibility_comparison
│   └── microservices_architecture_for_scalability
│       ├── service_discovery_mechanisms
│       └── state_management_in_microservices
└── ...
```

```
results
└────── cython_tensorflow_opencv_numpy_triton
  ├────── code_framework_function_package_file
  │  ├────── advanced_g4f_api_python_script
  │  │  ├────── advanced_g4f_api_python_script
  │  │  ├────── android_app
  │  │  ├────── update_moviepy_version
  │  │  ├────── wiki_page_image
  │  ├────── code_framework_function_package_file
  │  │  ├────── basic_text_input
  │  │  ├────── code_snippet
  │  │  ├────── parameter_change
  │  │  ├────── specific_function
  │  ├────── demoextractormaze_file
  │  │  ├────── claude3_folder
  │  │  ├────── demoextractormaze_file
  │  │  ├────── format
  │  │  ├────── png_mask
  │  └────── external_source_server
  │     ├────── incoming_packet
  │     ├────── new_module
  │     ├────── snapshot_management_implementation
  │     ├────── specific_server
  └────── cython_tensorflow_opencv_numpy_triton
     ├────── cloudflare_kv
     │  ├────── dpo
     │  ├────── finetune
     │  ├────── iry
     │  ├────── openwrt
     ├────── cosmos_discrete_tokenizer_codebook
     │  ├────── cosmos_discrete_tokenizer_codebook
     │  ├────── generative
     │  ├────── reformat
     │  ├────── unavailability
     ├────── java_enum
     │  ├────── cython_tensorflow_opencv_numpy_triton
     │  ├────── groq_api
     │  ├────── java_enum
     │  ├────── scrollbar
     └────── ------ other nodes ...
```

Taxogen
TaxoAdapt
TaxoCom

Figure 4: Illustrative snippets from the taxonomies generated by TaxoGen, TaxoAdapt, and TaxoCom.

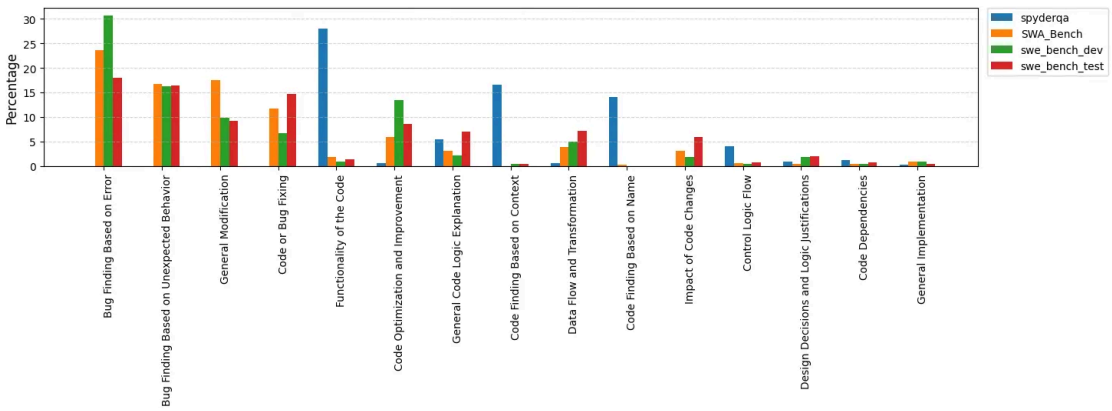

Figure 5: Comparison of diversity of different Software Engineering Benchmarks across the categories of our taxonomy.

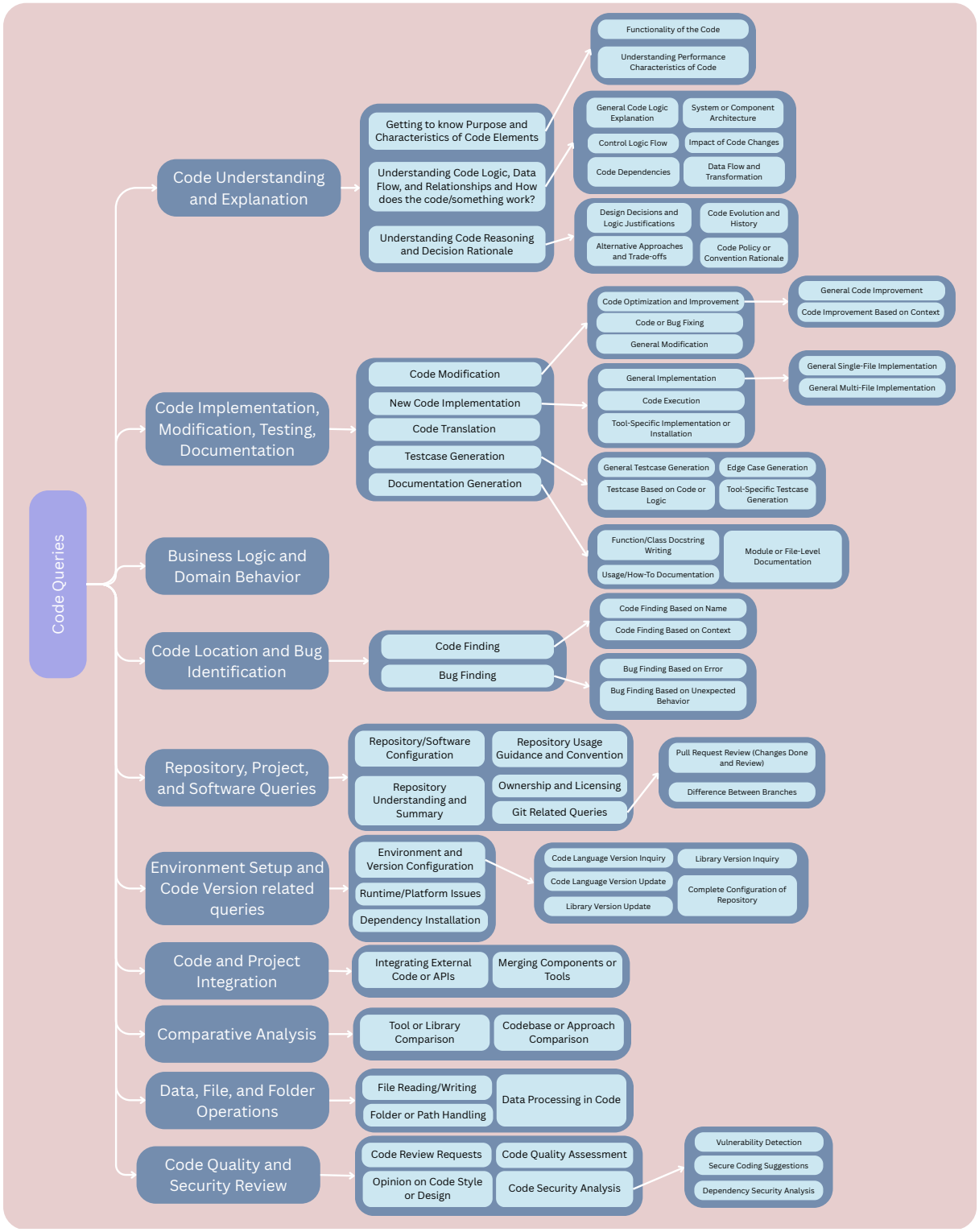

Figure 6: Final taxonomy after iterative refinement, with 10 top-level intent categories and 86 total nodes.