Towards Refining Developer Questions using LLM-Based Named Entity Recognition for Developer Chatroom Conversations

Pouya Fathollahzadeh, *Student Member, IEEE,* Mariam El Mezouar, Hao Li, Ying Zou, and Ahmed E. Hassan, *Fellow, IEEE*

***Abstract*—In software engineering chatrooms, communication is often hindered by imprecise questions that cannot be answered. Recognizing key entities (e.g., programming languages and libraries) and user intent (e.g., learning or requesting a review) can be essential for improving question clarity and facilitating better exchange. However, existing research using natural language processing techniques often overlooks these software-specific nuances. In this paper, we introduce SoftwarE-specific Named entity recognition, Intent detection, and Resolution classification (SENIR), a labelling approach that leverages a Large Language Model to annotate entities, intents, and resolution status in developer chatroom conversations. To offer quantitative guidance for improving question clarity and resolvability, we build a resolution prediction model that leverages SENIR's entity and intent labels along with additional predictive features. We evaluate SENIR on the DISCO dataset using a subset of annotated chatroom dialogues. SENIR achieves an 86% F-score for entity recognition, a 71% F-score for intent detection, and an 89% F-score for resolution status classification. Furthermore, our resolution prediction model, tested with various sampling strategies (random undersampling and oversampling with SMOTE) and evaluation methods (5-fold cross-validation, 10-fold cross-validation, and bootstrapping), demonstrates AUC values ranging from 0.7 to 0.8. Key factors influencing resolution include positive sentiment and entities such as `Programming Language` and `User Variable` across multiple intents, while diagnostic entities (e.g., `Error Name`) are more relevant in error-related questions. Moreover, resolution rates vary significantly by intent: questions about *API Usage* and *API Change* achieve higher resolution rates, whereas *Discrepancy* and *Review* have lower resolution rates. A Chi-Square analysis confirms the statistical significance of these differences.**

***Index Terms*—Empirical Software Engineering, Developer Chatroom, Name Entity Recognition, Large Language Models, Question Resolution, Mixtral**

I. Introduction

Developer chatrooms, such as Discord[1] and Slack,[2] are crucial tools for collaboration and knowledge sharing in software development. These platforms enable developers to seek help, solve technical problems, and engage continuously with the community. Collective knowledge available in chatrooms has been shown to accelerate software development and improve project outcomes [46], [52]. Additionally, chatrooms help build and maintain active developer communities, which are vital to the long-term success of open-source projects [50].

Despite the important role of chatrooms, their effectiveness is often hindered by communication issues. Poorly articulated questions that lack clarity or essential details frequently lead to misunderstandings, incomplete responses, or no response at all. For instance, research on Gitter[3] chatrooms finds that around 40% of questions remain unanswered [14] and delays in resolving problems discourage community participation [3]. While question-and-answer (Q&A) platforms like Stack Overflow are well studied [6], [29], [33], research on developer chatrooms is still limited. Initial findings suggest that including URLs can reduce response rates, whereas user mentions improve the rates [14]. Recent work by Lill et al. [32] has explored automated methods to improve developer support in chatrooms by identifying similar past conversations.

In developer chatrooms, a *conversation* consists of an initial question or statement (e.g., asking for help or seeking clarification) followed by responses, clarifications, and follow-ups until the question is resolved or left unresolved. The quality of the initial question impacts resolution outcomes, as well-formed questions often include sufficient technical details, such as code snippets or error types, making them easier to address. Named Entity Recognition (NER) is a useful method for extracting such technical details by identifying and categorizing domain-specific entities, such as libraries and error messages [31], [56]. While traditional NER methods work well on structured and formal text, they often require extensive feature engineering and task-specific training datasets, limiting their adaptability to informal and fragmented chatroom conversations. In contrast, Large Language Models (LLMs) are good at understanding complex natural language and generalizing across diverse contexts due to their pretraining on vast corpora.

To address these challenges, we propose an approach named as **S**oftwar**E**-specific **N**amed entity recognition, **I**ntent detection, and **R**esolution classification (SENIR) that leverages an LLM to label the chatroom conversations. SENIR recognizes software-specific entities, detects the intent behind a question in the context of developer chatrooms [24], and

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

Hao Li and Ahmed E. Hassan are with the Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen's University, Kingston, ON K7L 3N6, Canada. E-mail: hao.li@queensu.ca, ahmed@cs.queensu.ca.

Mariam El Mezouar is with the Department of Mathematics and Computer Science, Royal Military College of Canada, Kingston, ON K7K 7B4, Canada. E-mail: mariam.el-mezouar@rmc.ca.

[1]https://discord.com
[2]https://slack.com
[3]https://gitter.im

classifies whether a conversation has reached a satisfactory conclusion (i.e., resolved) or remains unresolved. This study evaluates SENIR using the DISCO dataset [47], a collection of 29,243 developer chatroom conversations extracted from various channels related to software engineering (SE) on Discord. Our work analyzes the developer chatroom conversations along the following three research questions (RQs):

RQ1: How effective is SENIR in labeling developer chatroom conversations?

Chatroom conversations are by nature informal, fast-paced, and lack sufficient context, making automated labelling a challenging task. To address this, we design SENIR using Mixtral 8x7B [27] to perform software-specific NER, intent detection, and resolution status classification. SENIR is evaluated on a manually labelled subset of 400 Discord conversations, achieving high accuracy (91% for NER, 76% for intent detection, and 93% for resolution status), with corresponding F-scores of 86%, 71%, and 89%.

RQ2: What features of the developer questions contribute to their resolution outcomes?

We use SENIR to automatically label 29,243 developer conversations from the DISCO dataset to identify software-specific entities, intents, and resolution status. The labelled entities and intents from questions are used to engineer features together with additional features such as sentiment, posting time, and question length. These question-related features are used to train a mixed-effect model to predict resolution outcomes. The model achieves an AUC of 0.75 and the feature importance analysis reveals that positive sentiment and technical specificity (e.g., use of library functions) positively impact resolution, while late posting times and excessive URLs negatively influence resolution.

RQ3: How do entities, intents, and their interactions impact the resolution of developer questions?

We analyze the 29,243 labelled conversations to examine how entities and intents influence resolution success. Chi-Square tests reveal significant differences in resolution rates across intents, with *API Usage* and *API Change* demonstrate better resolution rates (33.6% and 26.2%) than *Discrepancy* and *Review* (22.0%, and 18.8%). Within intents, specific entity pairs show better resolution outcomes than others. For instance, the pair (`Programming Language`, `Library Function`) is particularly effective for technical intents like *Errors* (63.6%), while (`UI ELEMENT`, `Website`) shows much lower success for intents like *Discrepancy* and *Review*.

The main contributions of our paper are as follows:

1) We propose SENIR, an LLM-based approach to automatically label developer chatroom conversations with software-specific named entities, intents, and resolution status.
2) We provide a predictive model for predicting resolution status based on the questions posted and reveal the most influential features for a question to be resolved.
3) By investigating how specific entities and user intents influence question resolution, we provide actionable insights to help developers refine their questions to potentially achieve higher response rates.
4) We release the annotated developer conversation dataset which is derived from DISCO and augmented with the SENIR labels in our replication package [17].

TABLE I
LIST OF CHANNELS IN DISCO DATASET

Channel	Number of Conversations
python#python-general	19,684
gophers#golang	8,860
racket#general	538
clojurians#clojure	161
Total	29,243

The rest of the paper is organized as follows: Section II outlines our methodology, including the case study design and data analysis. Section III presents the results for each research question. Section IV discusses the implications of our findings. Section V discusses related work. Section VI discusses the threats to validity. Section VII concludes the paper.

II. Case Study Design

This section presents our systematic approach for refining developer questions in chatrooms by leveraging an LLM. It includes details on data collection and preprocessing, our LLM-based approach for labelling developer chatroom conversations, manual labelling of a statistical sample set of conversations, and feature extraction. Fig. 1 presents an overview of our approach.

A. Data Collection and Preprocessing

To systematically analyze chatroom conversations, we first collect and preprocess data from the DISCO dataset [47], ensuring that our study focuses on relevant developer discussions. Table I presents an overview of the dataset.

Step 1: Extracting messages and metadata. We extract individual messages along with their associated conversation IDs, timestamps ("ts"), and user IDs from the DISCO dataset. For instance, a message like "How do I install X?" might include metadata such as "Mia" (the user who sent the message), "Aug 15, 2020, 10:34 AM" (the time it was sent), and "12345" (the conversation ID to which it belongs). Fig. 2 provides an example of a real conversation from the "racket#general" channel of Discord.

Step 2: Aggregating messages by conversation ID. Since each message is recorded separately without direct references to other messages, we group messages sharing the same "conversation ID" to reconstruct complete interactions. This aggregation enables further analysis (e.g., resolution status classification) at the conversation level rather than the individual message level.

Step 3: Augmenting conversations with additional metadata. To better understand when and over what period a conversation took place, we augment each conversation record with additional metadata. For instance, we calculate the range of months over which a conversation took place (e.g., "Aug2020–Oct2020") based on the start and end dates. This temporal information helps analyze patterns over time, such as

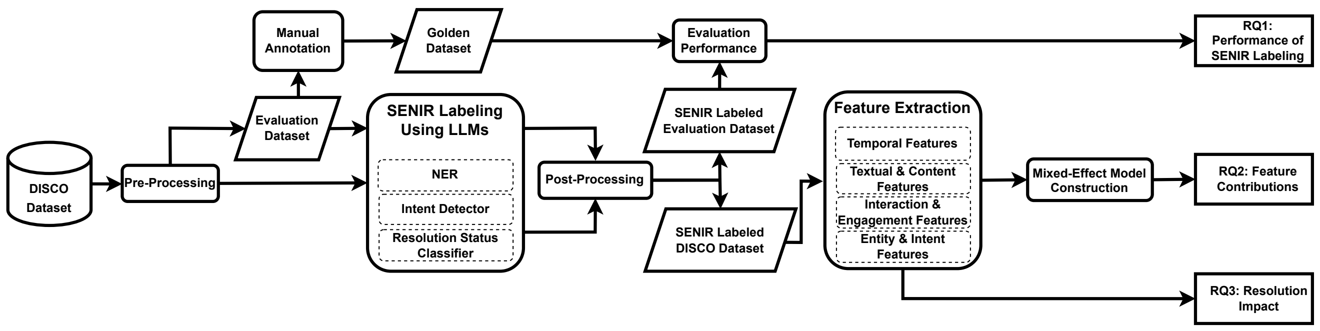
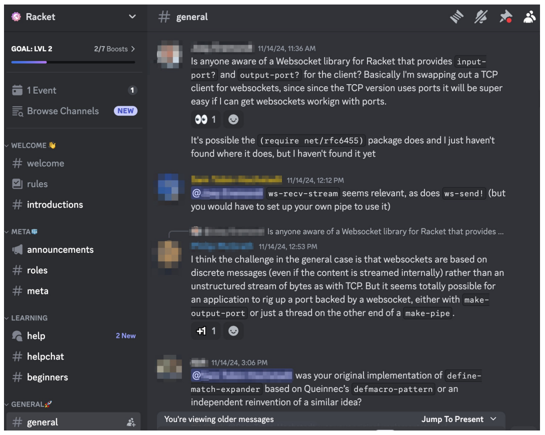

Manual Annotation
Golden Dataset
Evaluation Performance
RQ1: Performance of SENIR Labeling
DISCO Dataset
Pre-Processing
Evaluation Dataset
SENIR Labeling Using LLMs
NER
Intent Detector
Resolution Status Classifier
Post-Processing
SENIR Labeled Evaluation Dataset
SENIR Labeled DISCO Dataset
Feature Extraction
Temporal Features
Textual & Content Features
Interaction & Engagement Features
Entity & Intent Features
Mixed-Effect Model Construction
RQ2: Feature Contributions
RQ3: Resolution Impact

Fig. 1. Overview of our case study design.



[redacted] 11/14/24, 11:36 AM
Is anyone aware of a Websocket library for Racket that provides `input-port?` and `output-port?` for the client? Basically I'm swapping out a TCP client for websockets, since since the TCP version uses ports it will be super easy if I can get websockets workign with ports.

It's possible the `(require net/rfc6455)` package does and I just haven't found where it does, but I haven't found it yet

[redacted] 11/14/24, 12:12 PM
@[redacted] `ws-recv-stream` seems relevant, as does `ws-send!` (but you would have to set up your own pipe to use it)

[redacted] Is anyone aware of a Websocket library for Racket that provides ...
[redacted] 11/14/24, 12:53 PM
I think the challenge in the general case is that websockets are based on discrete messages (even if the content is streamed internally) rather than an unstructured stream of bytes as with TCP. But it seems totally possible for an application to rig up a port backed by a websocket, either with `make-output-port` or just a thread on the other end of a `make-pipe`.

[redacted] 11/14/24, 3:06 PM
@[redacted] was your original implementation of `define-match-expander` based on Queinnec's `defmacro-pattern` or an independent reinvention of a similar idea?

You're viewing older messages Jump To Present

Fig. 2. A conversation in the "racket#general" channel on Discord (The discussants' screen names are blurred for the purpose of privacy).

```
"team_domain": "Clojurians",
"channel_name": "clojure",
"month": "May-July 2020",
"messages": [{
  "conversation_id": "125",
  "messages": [{
    "msg_num": "1",
    "ts": "2020-05-03T08:14:12.575000",
    "user": "Ada",
    "text": "Hi, I'm new to programming. Any Python
        resources?"
  }, {
    "msg_num": "2",
    "ts": "2020-05-03T08:14:23.490000",
    "user": "Bob",
    "text": "We have a Python resources channel.
        @Shira might help."}]}]
```

Fig. 3. A sample JSON structure from the "clojurians#clojure" channel.

how long discussions tend to last or identify specific periods of activity.

Fig. 3 presents a sample JSON structure from the "clojurians#clojure" channel. It contains various metadata fields, including "team domain", "channel name", "month", "start date", and "end date", which provide context about where and when the conversations took place. Additionally, the "messages" field stores individual messages with attributes, such as "conversation ID", "message number", timestamp ("ts"), "user", and "text".

B. SENIR for Labelling Developer Chatroom Conversations

We design SENIR to label developer chatroom conversations, focusing on three tasks: **(1) Software-Specific NER:** NER identifies key software-related entities, such as `Library`, `Error Name`, and `Version`, which are crucial for understanding the context of developer discussions. **(2) Intent Detection:** Intent detection determines the purpose of each conversation, such as whether a user asks to learn a programming language or deals with an error. Understanding the intent of the initial question allows for better classification of conversations. **(3) Resolution Status Classification:** This classification task assesses whether a conversation has successfully resolved the initial question. Classifying resolved and unresolved questions helps evaluate the effectiveness of the chatroom and identifies questions requiring further attention. Throughout this paper, we follow a consistent notation for entities and intents: entities are written in `typewriter` (e.g., `Programming Language`), while intents are written in *italics* (e.g., *API Usage*).

Step 1: Selecting the LLM. Although SENIR is compatible with various LLMs, we select Mixtral 8x7B [27] due to its top performance and larger context window (33k tokens) compared to other open-source models on the chatbot arena leaderboard [9] as of January 23rd, 2024, when we started our research. The context window size refers to the maximum number of tokens that a model can process at one time when generating a response. A larger context window allows the model to take into account longer sequences of text, which is important when processing developer conversations that may contain up to 100 messages.

Step 2: Constructing prompts. To optimize performance, we design custom prompts to guide the LLM in handling specific tasks. The prompts include clear rules that instruct the model on what to focus on. For instance, in the NER prompt, the rules guide the LLM on which entities to extract (e.g., `Programming Language`). By following best practices in prompt engineering [25], [40], we enhance the model's ability to generate accurate and contextually relevant outputs.

NER prompt example: The model receives the initial question of a conversation along with timestamps, and is tasked with recognizing named entities. We collect a list of 28 software-specific entities based on prior work [13], [48], [54], [56] and include this pre-defined list to guide entity recognition. Table II provides examples and references for each of these entities.

TABLE II
SOFTWARE-SPECIFIC ENTITY CATEGORIES.

Entity	Examples	Reference
`Application`	Mosh, JKplayer, api-java-client	[13], [48], [54], [56]
`Programming Language`	Python, Java, CSS, C++	[13], [48], [54], [56]
`Version`	(Python) 2.7, (Windows) XP	[13], [48]
`Algorithm`	UDP, DFS, RBM	[48]
`Operation System`	Linux, iOS, Windows	[13], [48], [54], [56]
`Device`	Phone, Mobile, GPU	[13], [48], [54]
`Error Name`	Overflow, OutofRange, I/O Error	[48]
`User Name`	John, Maya, Clark, @Maya	[48], [54]
`Data Structure`	Array, List, Hash table, Heap	[48]
`Data Type`	String, Char, Double	[48]
`Library`	Numpy, Scipy, Auto-grad	[48], [54], [56]
`Library Class`	ItemTemplate, actionManager	[48], [54], [56]
`User Class`	myClass, TestClass	[13], [48]
`Library Variable`	math.inf, swing.color, ActionListener.Value	[48]
`User Variable`	user id, numberOfRowsInSection	[48]
`Library Function`	numpy.isinf(), Math.floor()	[48], [54], [56]
`User Function Name`	hello(), myFunction()	[48], [54], [56]
`File Type`	JSON, JAR	[48], [54], [56]
`File Name`	WindowsStoreProxy.xml, a.txt	[48]
`UI Element`	Button, Scroll bar, Text box	[48]
`Website`	MSDN, Google, Yahoo	[48]
`Organization`	Apache, Microsoft Research, Fair	[13], [48]
`License`	CC BY 4.0, Apache 2.0	[48], [56]
`HTML/XML Tag Name`	h1, div, img	[48], [54], [56]
`Value`	“hello world”, 255, 30.5, True	[48]
`In Line Code`	grep -rnw, select * from Tab	[48]
`Output Block`	Output from console/any IDE	[48]
`Keyboard Input`	CTRL+X, ALT, fn	[48]

Prompt: Extract all relevant software entities *[list of entities]* from the text below.
Input Format:
Question: I am using TensorFlow version 2.5 and encountering an error during installation. How can I fix this?
Output Format:
[“TensorFlow: Library”, “2.5: Version”]

Intent detection prompt example: The model receives the initial question of a conversation and is tasked with detecting the intent. A list of 7 categories of pre-defined intents is used to label the conversation with one or more intents. The 7 categories are collected based on previous work [1], [5]–[7], [43], [49] and Table III describes these intent categories.

Prompt: Identify the intents behind the question based on the following categories: [list of intents].
Input Format:
Question: How to install library X?
Output Format:
[“Learning”]

Resolution status classification prompt example: We also aim to use the LLM to classify whether a conversation is “resolved” or “unresolved” based on the content of the full conversation.

Prompt: Determine if the issue discussed in the following conversation is resolved.
Input Format:
- User 1: How do I fix this bug in Library Y?
- User 2: You should try reinstalling the dependency using version 3.0.
- User 1: That fixed the issue, thanks!

Output Format:
[“Resolved”]

C. Golden Dataset Collection and Manual Labelling

We construct a golden dataset of software engineering-related chatroom conversations to evaluate the performance of the LLM in automatically labelling software-specific named entities, intent, and resolution status of a conversation. To ensure a representative sample, we randomly select a statistically significant subset of the collected conversations (as listed in Table I) with a confidence level greater than 95% and a margin of error of 5%. This results in a selection of 400 conversations randomly chosen from the dataset. We manually verify each conversation to ensure that it begins with a software engineering-related question.

The manual labelling process involves annotating conversations with software-related entities, intents, and resolution statuses. Two annotators, both are PhD students (one is the first author of the paper) in software engineering, independently annotate the dataset to ensure a broad and unbiased perspective. The annotation process is conducted in two phases to ensure consistency and minimize bias:

Phase 1: Initial annotation and agreement check. The annotators first independently label a subset of 200 conversations, covering software-specific entities, intents, and resolution statuses. After that, we calculate inter-annotator agreement using Cohen’s Kappa [10] to measure consistency in labels. The initial Kappa scores are 0.71 for entity recognition (substantial agreement), 0.67 for intent detection (substantial agreement), and 0.76 for resolution status (substantial agreement). Disagreements, such as different interpretations of ambiguous intents like *Conceptual*, are discussed, and labelling guidelines are refined to improve clarity and ensure alignment (e.g., specifying cues for distinguishing *Learning* from *API Usage*).

Phase 2: Final annotation. Using the refined guidelines from Phase 1, the annotators label the remaining 200 conversations. The final Cohen’s Kappa scores across all 400 conversations are 0.84 for entity recognition (near-perfect

TABLE III
DESCRIPTIONS OF THE SEVEN INTENT CATEGORIES

Intent	Description	Reference
API Usage	This category subsumes questions of the types "How to implement something" and "Way of using something," as well as the categories "How-to" and "Interaction of API Classes." The posts in this category contain questions asking for suggestions on how to implement some functionality or how to use an API, with the questioner asking for concrete instructions.	[1], [5]–[7], [43], [49]
Discrepancy	This category contains questions about problems and unexpected behaviour of code snippets where the questioner has no clue how to solve them. It includes categories like "Do not work," "Discrepancy," "What is the Problem?" and "Why" (non-working code, errors, or unexpected behaviour).	[1], [6], [7], [43], [49]
Errors	Posts in this category deal with the problems of exceptions and errors, often including an exception and the stack trace. It is equivalent to "Error and Exception Handling" and overlaps with "Why" (non-working code, errors, or unexpected behaviour).	[5]–[7], [43], [49]
Review	This category merges "Decision Help" and "Review," "Better Solution," and "What" (concepts), as well as "How/Why something works" (understanding, reading, explaining, and checking). Questioners ask for better solutions or reviews of their code snippets, best practices, or decision-making assistance.	[1], [6], [7], [43], [49]
Conceptual	This category includes questions about the limitations of an API, API behaviour, understanding concepts such as design patterns or architectural styles, and background information about some API functionality. It is equivalent to "Conceptual" and includes "Why?" and "Is it possible?" as well as "What" and "How/Why something works."	[1], [6], [7], [43], [49]
API Change	This category concerns questions arising from changes in an API or compatibility issues between different versions. It includes "Version" and "API Changes."	[5]–[7]
Learning	This category merges "Learning a Language/Technology" and "Tutorials/-Documentation," where questioners seek documentation or tutorials to learn a tool or language on their own, rather than asking for specific solutions or instructions.	[5], [6], [49]

agreement), 0.78 for intent detection (substantial agreement), and 0.87 for resolution status (near-perfect agreement).

The manual labelling process consists of the following steps:

Step 1: Identifying software-related entities in the initial question of conversations. Each annotator manually identifies and labels software-related entities within the initial questions of conversations. For each identified entity, the annotator records the exact word representing the entity (e.g., `Library`) along with the timestamp of its occurrence in the conversation. This ensures precise tracking of where and when the entity appears in the discussion.

Step 2: Categorizing intents in the initial question of conversations. Each conversation is labelled based on its initial question with one or more intents that indicate its underlying purpose. The intent annotation follows pre-defined categories (as shown in Table III) to maintain consistency.

Step 3: Annotating resolution status for conversations. Unlike Stack Overflow, Discord or similar chatrooms do not have a flag to indicate whether a question is resolved. To address this, each annotator manually labels the resolution status of each conversation. A conversation is labelled as "resolved" if the original question receives a relevant answer, and as "unresolved" if it does not.

D. Feature Extraction

Feature extraction helps analyze and understand whether a conversation in the DISCO dataset can be resolved or not when the initial question is posted. The extracted features capture key aspects of the initial question, such as timing, content quality, interaction levels, and the presence of specific entities and intents. By linking these features to resolution outcomes, we gain insights into what makes a question more likely to be resolved. Feature extraction follows two steps:

Step 1: Labelling the DISCO dataset using SENIR. We run SENIR on the entire dataset (illustrated in Fig. 1) which consists of 29,243 conversations to label the entities, intents, and resolution statuses for each conversation.

Step 2: Extracting features from the labelled DISCO dataset. We engineer and extract a set of features from the labelled dataset and categorize them as temporal features, textual & content features, interaction & engagement features, and entity & intent-related features. Each category and its related features are listed in Table IV and described below. Except for entity & intent-related features, all extracted features are collected from previous work [2], [4], [8], [12], [14], [19], [26], [36], [45], [53], [55].

Temporal features. These features capture the timing aspects of the initial question, based on the hypothesis that posting time may influence resolution likelihood. Specifically, we include "weekday", representing the day of the week the question was asked, and "daytime", indicating the hour of the day the question was posted (ranging from 0 to 23 hours).

Textual & content features. This set of features is related to the structure and content quality of the initial question. The goal is to assess whether question clarity and readability impact resolution, as these can directly affect user engagement and the likelihood of achieving a resolution. For example, a higher readability score may indicate that the question is easier to understand. Features include "URLs count", capturing the number of URLs present in the question, and "code snippets", indicating whether the question contains code blocks.

Interaction & engagement features. These features focus on the questioner's activity, capturing how interaction and engagement may influence resolution likelihood. Examples include "active chatroom questioner", which identifies whether the questioner is an active participant in the chatroom (binary: 0 for inactive, 1 for active), and "questioner received response ratio", which measures the proportion of the questioner's past questions that have received responses.

Entity & intent-related features. These features leverage the labelled outputs from SENIR to provide insights into

TABLE IV
THE LIST OF FEATURES USED IN THE STUDY AND THEIR DESCRIPTIONS.

Category	Feature	Description	Reference
Temporal Features	Weekday	Day of the week the question took place	[2], [14]
	Daytime	Time of day the question occurred	[14]
Textual & Content Features	Readability CLI	Readability score of the question text	[12], [14], [55]
	Text-Code Ratio Question	Ratio of code snippets to regular text	[36]
	URLs Count	Number of URLs mentioned	[14], [53]
	User Mentions	Number of times users are mentioned	[4], [53]
	Code Snippets	Presence of code blocks or snippets	[14], [55]
	Question Length	Length of the initial question	[8]
Interaction & Engagement Features	Sentiment	Sentiment of the initial question	[19], [26]
	Active Chatroom Questioner	Indicator of whether the questioner is an active member	[14], [36], [45]
	Questioner Received Response Ratio	Ratio of received responses to total questions asked	[36], [45]
Entity & Intent-Related Features	Total Entities Count	Total number of entities recognized	
	Unique Entities Count	Number of unique entities	
	Entity Occurrences	Frequency of each entity within the question	
	Intent Total Count	Total count of identified intents	
	List of 7 Intents	Specific intent categories	
	List of 28 Entities	Specific software-related entities	

the technical aspects of the initial question. For example, "total entities count" represents the total number of recognized entities in the question and "unique entities count" shows the number of distinct entities present. Additionally, "entity occurrences" measures how frequently entities appear within the question.

III. Results

In this section, we provide the motivation, approach, and findings for each of our research questions.

A. RQ1: How effective is SENIR in labeling developer chatroom conversations?

Motivation. Annotating chatroom conversations with relevant entities and intents can enhance problem-solving and knowledge sharing in SE communities like Discord. Precisely identifying key technical details can lead to higher-quality responses. However, existing labelling approaches face significant challenges. Traditional NER and intent detection methods struggle with the informal, fragmented nature of developer chatrooms, where multiple topics often overlap. Manual labelling, while effective, is resource-intensive and limits scalability.

To address these challenges, we explore the use of the LLM to automate the labelling of developer chatroom conversations. With strong contextual awareness, the LLM chosen by our work may be able to better navigate informal discussions compared to traditional methods. In this RQ, we evaluate the effectiveness of the LLM in automatically labelling software-related chatroom conversations. Specifically, we introduce SENIR, an approach that leverages the LLM to detect and label software-specific entities, user intents, and resolution status. The labelled dataset provides a basis for further investigation in RQ2 and RQ3.

Approach. We evaluate the labelling performance of SENIR using the golden dataset (see Section II-C), which contains 400 manually annotated developer conversations. We compare labelling output from SENIR with manual labels using the performance metrics *Precision*, *Recall*, *F-score*, and *Accuracy*. *Precision* measures the proportion of correctly predicted positive instances out of all predicted positives, while *Recall* measures the proportion of true positives out of all actual positive instances. The *F-score* represents the harmonic mean of *Precision* and *Recall*, balancing both metrics. *Accuracy*, on the other hand, is the proportion of all correct predictions (both true positives and true negatives) to the total number of predictions [21].

TABLE V
PERFORMANCE METRICS FOR ENTITY RECOGNITION, INTENT IDENTIFICATION, AND RESOLUTION STATUS DETECTION.

Metric	Entity (%)	Intent (%)	Resolution (%)
Accuracy	91	76	93
Precision	88	72	96
Recall	85	70	87
F-Score	86	71	89

To calculate the evaluation metrics, we map SENIR's predictions and the golden labels into the confusion matrix categories: *True Positive* (TP), *False Positive* (FP), *False Negative* (FN), and *True Negative* (TN).

The evaluation metrics are calculated as follows:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{1}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2}$$

$$\text{F-Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3}$$

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + FN + TN} \tag{4}$$

The definition of each confusion matrix category depends on the specific labelling task:

(1) Entity Recognition (Token-Level)

- **Golden labels:** For each token in the initial question of a conversation, the golden dataset indicates either a specific entity type (e.g., `Programming Language`, `Error Name`) or non-entity.
- **Predicted labels:** SENIR assigns an entity type or non-entity to each token.
- **TP:** SENIR correctly labels a token with the same entity type as in the golden dataset.
- **FP:** SENIR assigns an entity type to a token that is labelled as non-entity or a different entity type in the golden dataset.
- **FN:** SENIR fails to assign an entity type to a token that should have one.
- **TN:** SENIR correctly labels a token as non-entity (or correctly refrains from assigning a wrong entity type).

(2) Intent Detection (Conversation-Level, Multi-Label)

- **Golden labels:** Each conversation in the golden dataset may have one or more intent labels (e.g., *API Usage*, *Review*, *Discrepancy*) based on the initial question.
- **Predicted labels:** SENIR outputs a set of intents for each conversation.
- **TP:** An intent predicted by SENIR matches one in the golden labels.
- **FP:** An intent predicted by SENIR does not appear in the golden labels.
- **FN:** An intent present in the golden labels is not detected by SENIR.
- **TN:** An intent is correctly not predicted (i.e., it appears in neither SENIR's output nor the golden labels).

(3) Resolution Status (Conversation-Level, Single-Label)

- **Golden label:** Each conversation is labelled with a single resolution outcome (i.e., "resolved" or "unresolved").
- **Predicted label:** SENIR outputs exactly one resolution outcome for each conversation.
- **TP:** SENIR correctly predicts the outcome as "resolved" when the golden label is "resolved".
- **FP:** SENIR predicts "resolved" when the golden label is "unresolved".
- **FN:** SENIR predicts "unresolved" when the golden label is "resolved".
- **TN:** SENIR correctly predicts "unresolved" when the golden label is "unresolved".

***Results.* SENIR can correctly label 91% of entities.** Table V summarizes of the performance metrics for each task, SENIR achieves strong performance in labelling resolution status with an F-score of 89%. For the NER and intent detection tasks, SENIR also performs well with F-scores of 86% and 71%, respectively. Although the recall values are slightly lower, the precision and accuracy metrics remain robust, which indicates that SENIR can accurately capture the software-specific entities and intents in developer conversations.

Our LLM-based approach shows strong performance for concrete entities (e.g., `Error Name`, `Library Function`), but lower performance for abstract ones (e.g., `Application`). Concrete entities, such as `Error Name` (94% accuracy) and `Library Function` entities (88% accuracy), are easier to detect likely due to their well-defined context. In contrast, abstract terms like `Application` achieve a lower accuracy of 81%. These results suggest that entities grounded in concrete, well-defined terms are easier for the model to identify compared to abstract concepts. This observation aligns with previous findings by Ye et al. [56], where the baseline approach also performs well on programming languages but struggles with more nuanced categories such as the "API category".

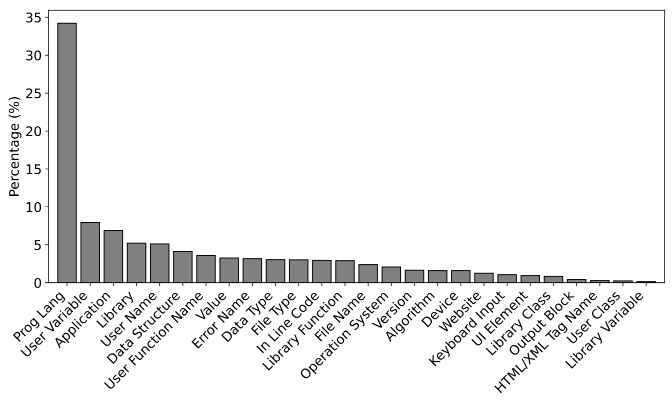


Fig. 4. Distribution of SENIR-labelled entities in the DISCO dataset.

Entities related to structured data (e.g., `Version` and `File Type`) consistently achieve high precision and recall. Entities that represent structured information, such as `Version` and `File Type`, consistently rank the highest in terms of precision and recall (as shown in Fig. 4). In contrast, entities such as `Library` and `In Line Code` show lower performance, which indicates that these categories may be more ambiguous or challenging for the model to detect in informal chatroom conversations.

`Programming Language` is the most frequently recognized entity in developer conversations. Fig. 4 shows the distribution of recognized entities, with `Programming Language` appearing in 34% of the conversations. This finding highlights the centrality of discussions surrounding programming languages in developer chatrooms. `User Variable` and `Application` are the next most frequent entities, indicating a significant focus on user-defined variables and software applications.

Intent detection shows varied performance, indicating differing levels of contextual complexity among intents. Table V shows that certain intents, such as *Review* and *Errors*, achieve high accuracy (89%), compared to more abstract intents, such as *Conceptual* (57% accuracy). This discrepancy suggests that intents that require deep contextual understanding are more challenging for the model to capture accurately. In some instances, errors occur when entities or intents are used in unexpected contexts. For example, the term 'Go' is misclassified in discussions about travel rather than programming. This highlights a potential limitation in the model's ability to disambiguate between terms with multiple interpretations, particularly in casual or off-topic conversations.

Developer conversations predominantly focus on learning new concepts and understanding complex ideas. Fig. 5 shows that *Learning* and *Conceptual* are the most common

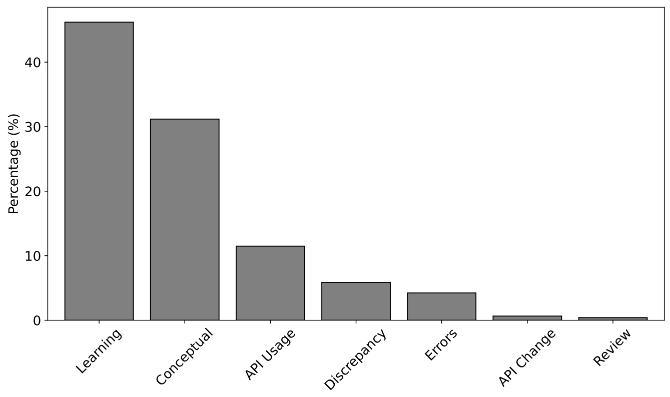


Fig. 5. Distribution of SENIR-labelled intents in the DISCO dataset.

intents in the dataset, representing 46% and 31% of all conversations. This reflects the focus of technical discussions on technical education and conceptual exploration. In contrast, *API Usage*, the third most frequent intent (11%) highlights the need for clarifications or advice on API functionalities.

Our LLM-based labelling approach shows strong agreement with manual labels. Our approach achieves Cohen's Kappa scores of 0.84 for entity recognition, 0.78 for intent detection, and 0.86 for resolution status. These scores fall within the “substantial” (0.61–0.80) and the “perfect” (0.81–1.00) agreement [34], indicating that SENIR reliably captures key aspects of conversations. However, instances of disagreement between annotators and the model often arise due to conversational ambiguity. As the example shown below, certain conversations could be labelled as either *Learning* or *API Usage*, leading to different labels. Disagreement also occurs when users use vague terminology in a particular development community. These challenges highlight the difficulty of intent detection in casual, context-shifting environments like developer chatrooms.

- User 1: Hey folks, I'm confused about when to use async/await over threading in Python. Any insights?
- User 2: Async is for concurrency, great for I/O-bound tasks. Threading is more about parallelism.
- User 3: What's your use case?
- User 1: I have a function that fetches data from multiple APIs. Should I go with async or just use threads?
- User 4: If it's API calls, async is usually better. Use ‘asyncio‘ and ‘aiohttp‘ for handling multiple requests.
- User 1: Got it! I'll try that. Thanks!

Summary of RQ1

SENIR demonstrates high effectiveness in analyzing developer chatroom conversations by accurately identifying software-specific entities, intents, and resolution statuses, with an F-score of 86% for entity recognition and 89% for resolution status classification. Despite these strengths, challenges remain in detecting certain intents requiring deep contextual understanding, particularly for abstract concepts, highlighting areas for future improvement.

B. RQ2: What features of the developer questions contribute to their resolution outcomes?

Motivation. In RQ1, we propose an LLM-based approach to label chatroom conversations with software-specific entities, the intent of the question, and the resolution outcome of the conversation. Building on this, we apply our approach to label a large dataset of 29,243 conversations sourced from Discord. This labelled dataset is used to further analyze the relationship between the characteristics of questions and their associated resolution outcomes. The goal of RQ2 is to investigate which features of developer questions most significantly influence their likelihood of being resolved. Our goal is to gain insights into the factors that contribute to successful resolutions, as well as those that may harm them, by training a model on the question-derived features.

Approach. To address RQ2, we follow the steps listed below:

1. Labelling the Conversations: We use the approach from RQ1 to label 29,243 conversations sourced from four distinct Discord channels within the DISCO dataset (see Table I). Each conversation is annotated with software-specific entities, the intent of the question, and the resolution outcome. The resolution outcome is used as the dependent variable in building the prediction model.

2. Feature Extraction: For each conversation, we isolate the initiating question and extract a comprehensive set of features from the initial questions, as described in Section II-D. These features include general attributes, such as question length and sentiment, as well as features derived from the labelling in Step 1, such as entities count in a given question. Notably, the features pertain solely to the question itself. The model is not trained on any features irrelevant to the initial question as a whole. The feature set consists of 50 question-related features, which are listed in Table IV.

3. Feature Processing: We first apply max-min normalization to scale all features to a range of [0, 1]. Without normalization, features with larger ranges (e.g., [-100, 100]) could disproportionately influence the model's performance compared to features with smaller ranges (e.g., [0, 10]). By transforming all features into the same range, we avoid potential biases during model training. The normalization formula is as follows:

$$X_{\text{Normalized Value}} = \frac{X - X_{\min}}{X_{\max} - X_{\min}} \in [0, 1]$$

Where $X_{\min}$ and $X_{\max}$ are the minimum and maximum values of the feature, respectively.

After normalization, we perform a correlation and redundancy analysis to identify the highly correlated and redundant features.

Correlation refers to the degree to which two features are related. Highly correlated features provide similar information to the model, which can introduce instability in model coefficients and reduce interpretability. We use Spearman's correlation coefficient to measure the degree of correlation between features. Following the previous studies [37]–[39], we consider a correlation coefficient threshold of 0.7 for the correlation coefficient values to identify strong correlations.

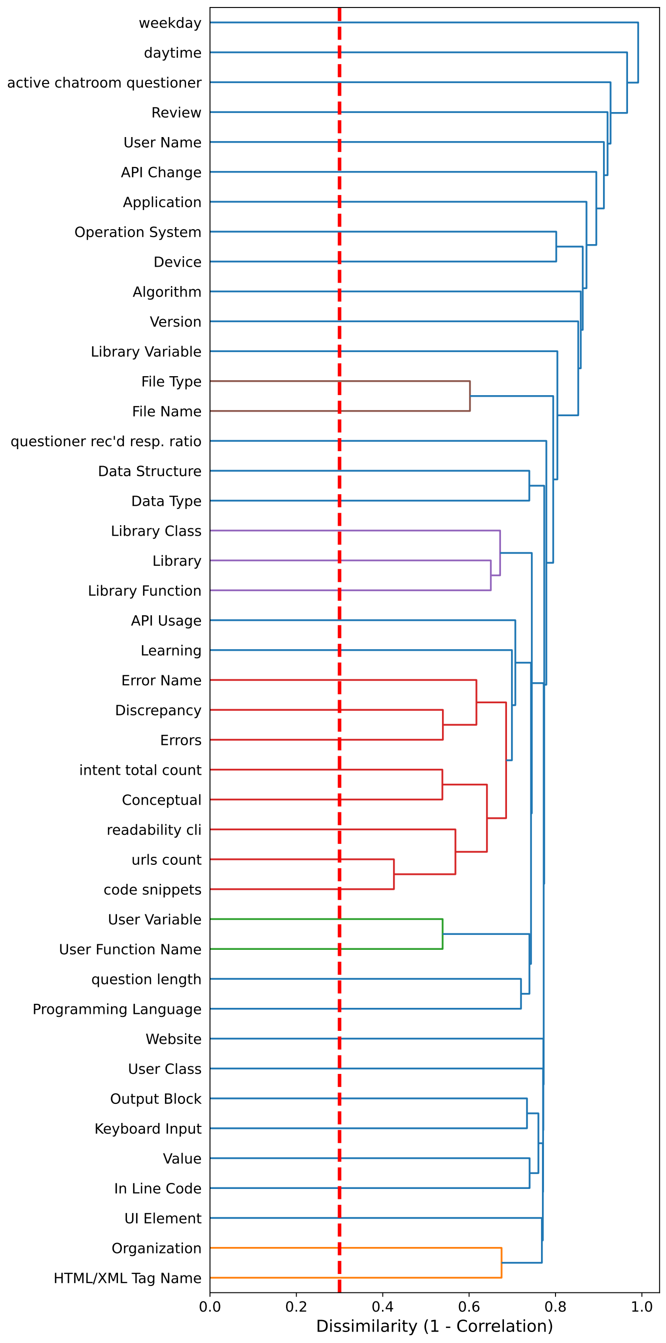


Fig. 6. Features correlation matrix dendrogram.

To visualize the correlation structure, we apply hierarchical clustering, which requires a distance metric. We transform the correlation values into a dissimilarity measure defined as:

$$\text{distance} = 1 - \text{correlation} \tag{5}$$

In this transformed space, highly correlated feature pairs (correlation 0.7) have distances 0.3. We therefore set a cut-off threshold of 0.3 on the (Fig. 6) x-axis of the dendrogram and randomly remove one feature from each highly correlated pair to ensure that we retain a diverse and representative set of features.

Redundancy occurs when a feature does not provide unique information and can be predicted using other features. To detect redundancy, we compute the Variance Inflation Factor (VIF), a widely used metric that quantifies how much a given feature is explained by other independent variables. Following prior work [28], [41], we adopt a cut-off threshold of 10, where VIF values above this level indicate substantial redundancy. Features exceeding this threshold are excluded to improve computational efficiency and model generalizability.

After applying both correlation and redundancy filtering, we remove 15 highly correlated and redundant features: Text-Code Ratio Question, User Mentions, Total Entities Count, Unique Entities Count, Entity Occurrences, Sentiment, `License` Intent Total Count, *API Usage*, *Conceptual*, *Discrepancy*, *Errors*, *Review*, *API Chance*, and *Learning*. This process reduces the feature space from 50 to 35 features while maintaining key predictive information.

4. Model Training: The dataset consists of a total of 29,182 questions, with 22,748 labelled as “unresolved” (77.95%) and 6,434 labelled as “resolved” (22.05%). This class imbalance is addressed through sampling strategies during model training. To account for the variability of data sourced from different chatrooms, we train a mixed-effect model that incorporates: **(i)** fixed effects, which capture the influence of extracted features, and **(ii)** random effects, which address variations across chatrooms. To refine the model and prevent overfitting, we employ the stepwise regression algorithm [58]. Starting with an empty model, features are iteratively added based on their contribution to predictive performance. Specifically, we employ a stepwise feature selection approach, where features are evaluated based on their impact on performance metrics, such as the Akaike Information Criterion (AIC). Features that significantly improve model fit and predictive power are retained, while those with minimal or redundant contributions are excluded. This process reduces the initial feature set to 20 impactful features, which are detailed in Table VII.

Given the class imbalance, we explore two sampling strategies during training: **(i)** Random undersampling, where the majority class is reduced to match the minority class, and **(ii)** Oversampling using the Synthetic Minority Oversampling Technique (SMOTE), which generates synthetic samples for the minority class.

5. LLM Baseline (Prompt-Based Classification): As a baseline for resolution prediction, we also prompt a general-purpose LLM, specifically Mixtral 8x7B [27]. In a zero-shot setting, we provide only the initial question to the LLM and ask it to predict whether the question is likely to be “resolved” or “unresolved”. An example of the prompt is shown below.

Prompt: You are an AI assistant who helps analyze software engineering chatroom discussions. Your task is to determine whether the following developer question will likely receive a resolved answer based on its clarity, completeness, and technical details.
Instructions:
- Read the developer question carefully.
- Predict whether the question will be resolved (*Yes*) or not resolved (*No*).
- Provide a confidence score between 0% and 100%, indicating how sure you are about your prediction.

TABLE VI
AUC VALUES OF THE MIXED-EFFECT MODELS FOR THE DIFFERENT CONFIGURATIONS

Configuration	5-Fold CV	10-Fold CV	Bootstrap
Random Undersampling	0.7544	0.7545	0.7531
Oversampling with SMOTE	0.7560	0.7558	0.7546

- Do not provide an explanation—only return the classification and confidence score.

Input Format:
Question: I am using TensorFlow version 2.5 and encountering an error during installation. How can I fix this?
Output Format (Strict JSON format):
{"resolution_status": "Yes", "confidence_score": "82%"}

We then compare these baseline predictions (derived solely from the initial question) to the ground-truth resolution labels, which are obtained via an LLM-based method that has access to the *entire* conversation (as explained in RQ1).

6. Model Evaluation: The trained mixed-effect model is evaluated using the **Area Under the Curve (AUC)**, which measures the model's ability to distinguish between resolved and unresolved questions. To ensure a stable evaluation, we use the following approaches: **(i) 5-Fold Cross-Validation (CV)**, where the dataset is split into 5 folds, and each fold is used as a test set once; **(ii) 10-Fold Cross-Validation (CV)**, which follows the same approach except with 10 folds; and **(iii) Bootstrapping (100 iterations)**, where the dataset is repeatedly resampled with replacement to capture variability. Each evaluation method is applied across both sampling strategies (random undersampling and SMOTE).

In parallel, we also evaluate the LLM's baseline predictions to obtain an AUC value used for direct comparison.

7. Feature Importance Analysis: We conduct a feature importance analysis to identify which features have significant positive or negative impacts on the likelihood of resolution. We examine the coefficients from both the final model and the intermediary models at each step of the feature selection process. By analyzing the changes in coefficients as new features are added, we assess how each feature contributes to the model incrementally. Positive coefficients indicate features that increase the likelihood of resolution, while negative coefficients indicate those that decrease it. This stepwise examination ensures that the final set of features is robust and their impact remains consistent throughout the process.

To assess the statistical significance of these coefficients, we examine the associated z-values and p-values. The z-value measures how many standard deviations the coefficient is from zero under the null hypothesis (H_0: *The feature does not affect the likelihood of question resolution*), with higher z-values indicating stronger evidence against the null hypothesis. Features with $p < 0.05$ are considered statistically significant.

Results. **The model demonstrates acceptable discrimination performance across all configurations, with AUC values consistently falling within the range of 0.7 to 0.8.** Table VI summarizes the AUC values for the tested configurations, which include two sampling strategies (random undersampling and oversampling with SMOTE) evaluated using three methodologies: 5-Fold CV, 10-Fold CV, and Bootstrapping. Oversampling with SMOTE has a slight advantage and achieves the highest AUC values across all evaluation approaches, with an AUC of 0.7560 in 5-Fold CV and 0.7558 in 10-Fold CV. The stable results across all configurations provide confidence in the model's ability to distinguish between resolved and unresolved questions, allowing us to proceed with analyzing feature importance to understand the factors contributing to resolution outcomes.

TABLE VII
MIXED-EFFECT MODEL RESULTS FOR RESOLUTION PREDICTION

Feature	Coef.	Std.Err.	z	P>|z|	Sign.	Rel.
Sentiment	0.705	0.044	15.912	0.000	***	↗
`User Name`	0.097	0.017	5.745	0.000	***	↗
URLs Count	-0.298	0.077	-3.846	0.000	***	↘
`Application`	-0.033	0.015	-2.265	0.024	*	↘
`Library`	-0.062	0.018	-3.472	0.001	**	↘
Daytime	-0.020	0.014	-1.475	0.140		↘
`Library Function`	0.065	0.022	2.931	0.003	**	↗
Weekday	0.018	0.012	1.482	0.138		↗
`UI Element`	-0.061	0.043	-1.415	0.157		↘
Code Snippets	-0.380	0.103	-3.705	0.000	***	↘
`Data Type`	0.036	0.021	1.707	0.088	.	↗
`Data Structure`	0.042	0.018	2.350	0.019	*	↗
Readability CLI	-0.347	0.216	-1.605	0.109		↘
`File Type`	-0.42	0.021	-1.941	0.052	.	↘
`Keyboard Input`	0.84	0.037	2.241	0.025	*	↗
`Library Class`	0.86	0.040	2.154	0.031	*	↗
Question Length	-0.124	0.072	-1.716	0.086	.	↘
`Organization`	-0.167	0.083	-2.012	0.044	*	↘
Specific Intent Presence	-0.115	0.016	-6.976	0.000	***	↘
`HTML/XML Tag Name`	0.124	0.069	1.782	0.075	.	↗

Furthermore, when comparing these results to the LLM baseline, we observe that the AUC values for the LLM across different confidence scores remain consistently low, ranging from 0.50 to 0.53, indicating that its predictions are akin to random guessing. We conclude that the performance of the baseline LLM falls well below the 0.70 to 0.80 range achieved by the feature-based mixed effect model.

While certain SE entities provide helpful context to drive resolution, others may introduce unnecessary complexity or ambiguity. The subset of features representing SE entities—denoted in capital letters—provides insights into how the presence of specific entities influences question resolution, as shown in Table VII. These features are binary, indicating whether a particular entity is present (1) or absent (0) in a question. Among these, `Application` (e.g., "Flask," "PyCharm"), `Library` (e.g., "NumPy," "React"), and Code Snippets have a significant negative effect on resolution likelihood ($p < 0.05$). For instance, questions that include Code Snippets are negatively correlated with resolution ($z = -3.705$, coefficient=-0.380), potentially indicating that such questions might introduce complexity or require additional context that impedes resolution. Similarly, the presence of `Application` and `Library` entities appears to slightly detract from resolution outcomes, possibly due to their broad or ambiguous nature. In contrast, entities like `Library Function` (e.g., "numpy.mean()," "json.dumps()") and `Library Class` (e.g., "DataFrame," "Button") demonstrate a positive correlation with resolution

($p < 0.05$), suggesting that including specific, well-defined technical elements increases the likelihood of resolution. For example, the feature `Library Function` has a coefficient of 0.065 ($z = 2.931$), indicating a modest but significant positive relationship. This finding highlights the value of technical specificity in driving resolution success.

Other features with a positive impact include Sentiment, where a more positive tone significantly improves resolution likelihood, and User Name, indicating that directly mentioning specific users could improve responsiveness. This result confirms the findings by Ehsan et al. [14]. On the other hand, URLs Count has a negative effect on resolution, indicating that later excessive URLs may introduce complexity or reduce focus, making questions harder to address effectively.

Summary of RQ2

The mixed-effect model demonstrates acceptable performance in distinguishing resolved and unresolved questions, with stable AUC values across all configurations (0.7 to 0.8). Feature importance analysis highlights that while specific SE entities (e.g., `Library Function`, `Library Class`) and positive sentiment improve resolution likelihood, elements like excessive URLs, and entity mentions detract from resolution.

C. RQ3: How do entities, intents, and their interactions impact the resolution of developer questions?

Motivation. The quality and formulation of developer questions are critical for improving response rates and overall quality of interactions. In RQ3, we aim to understand how specific combinations of software-specific entities and intents influence the question resolution. By providing insights into which entities and intents lead to higher resolution rates, our goal is to guide developers in crafting more effective questions, thereby reducing the number of unanswered questions.

Approach. We study the interaction among software-specific entities, intents, and the resolution of developer questions through the following aspects:

1. Intent Success Rate Evaluation: To understand the overall efficacy of different question types in eliciting responses, we assess the success rate for each intent by calculating the percentage of resolved questions within each intent category. We investigate if there is a significant relationship between the intent of the question and its resolution status. To evaluate this, we use a Chi-Square test of independence [35], which determines if there is an association between two categorical variables. The test compares the observed frequencies (the actual number of resolved and unresolved questions per intent) to the expected frequencies (what would be expected if there were no relationship). A large Chi-Square statistic indicates that the observed and expected values differ significantly, suggesting a relationship between the variables. Conversely, a small Chi-Square statistic suggests little or no relationship. The significance of the test is evaluated by the p-value, where a small p-value (typically less than 0.05) indicates that the observed relationship is unlikely due to chance.

TABLE VIII
RESOLUTION OUTCOME BY INTENT

Intent	% Success	# Success	# Total
API Usage	33.6	1,845	5,497
API Change	26.2	81	309
Errors	25.6	519	2,024
Conceptual	23.7	3,535	14,924
Learning	22.9	5,053	22,112
Discrepancy	22.0	618	2,813
Review	18.8	36	192
Overall	21.9	6,412	29,243

2. Analysis of Entity Pairs: We analyze entity pairs, defined as two entities appearing together in the initial question of a conversation. For questions involving three or more entities, all possible pairs are considered. For example, if a question contains the entities `Device`, `Application`, and `Library`, the resulting pairs are (`Device`, `Application`), (`Device`, `Library`), and (`Application`, `Library`). This analysis is motivated by our observation that around 75% of conversations involve at least two entities. The goal of this is to understand how these relationships contribute to question resolution. In addition, we examine the resolution success rates of entity pairs across different intent categories in their effectiveness. To ensure reliable results, we filter out pairs with fewer than 10 occurrences.

Results. **The success rates for the different intents vary considerably from 18.8% to 33.6%, highlighting an overall suboptimal rate of resolution across all intents.** Table VIII shows that *API Usage* (33.6%) and *API Change* (26.2%) have the highest success rates, while *Discrepancy* (22%) and *Review* (18.8%) are on the lower end. This points to potential difficulties in addressing more complex or nuanced questions. To further assess the relationship between intent and resolution status, a Chi-Square test is conducted. The test yields a Chi-Square statistic of 76.83 with 6 degrees of freedom and a p-value of $1.61e^{-14}$. The large Chi-Square statistic, far exceeding the critical threshold, and the small p-value strongly indicate that the differences in resolution rates across intents are statistically significant. This suggests that certain intents are more likely to result in successful resolution than others, reinforcing the need for better question formulation in lower-performing categories like *Discrepancy* and *Review*.

Entity pair analysis reveals that certain combinations are more effective in driving question resolution under specific intents. As shown in Table IX, for the *API Change* intent, the pair (`Data Structure`, `Output Block`) achieves a success rate of 75%, On the other hand, the pair (`File Type`, `Keyboard Input`) yields 17.7% suggesting these entities might be less useful in eliciting responses.

For the *API Usage* intent, the pair (`Application`, `File Type`) has a success rate of 53.3%, showing that providing specific application-related and file-related information is beneficial. However, combinations involving the pair (`Value`, `Keyboard Input`) yield only 5.6%, indicating their ineffectiveness in this context.

In the *Learning* intent, the pair (`Programming Language`, `Library`) has a success rate of 53.9%,

TABLE IX
ENTITY PAIR RESULTS (TOP 3 AND BOTTOM 3 BY SUCCESS RATE).

Intent	Entity Pair	% Succ	# Total
Aggregate Results (All Intents)	`Device, Library Variable`	45.5	11
	`Version, Library Variable`	41.7	12
	`Data Type, Library Variable`	36.4	22
	`Application, User Class`	10.0	20
	`User Variable, UI Element`	8.4	154
	`User Class, Value`	6.9	29
The following rows present entity pair results broken down by intent.			
API Usage	`Application, File Type`	53.3	15
	`Operation System, User Variable`	45.8	24
	`Data Structure, Library Function`	45.1	51
	`Value, Keyboard Input`	5.6	18
	`User Func Name, Keyboard Input`	5.0	20
	`Value, Output Block`	0.0	10
Conceptual	`Data Structure, Library Variable`	42.1	19
	`User Name, HTML/XML Tag Name`	41.7	12
	`Version, Library Variable`	41.7	12
	`User Func Name, Website`	6.9	29
	`Algorithm, UI Element`	6.3	16
	`User Variable, Website`	5.6	36
Discrepancy	`User Variable, Library Function`	39.1	115
	`Library Class, Library Function`	38.7	31
	`Application, File Type`	38.5	26
	`Data Type, Keyboard Input`	0.0	10
	`File Name, UI Element`	0.0	11
	`User Func Name, Keyboard Input`	0.0	21
Errors	`Prog Lang, Library Function`	63.6	11
	`Prog Lang, File Name`	41.2	17
	`Prog Lang, File Type`	33.3	12
	`Prog Lang, Error Name`	27.3	11
	`Prog Lang, Version`	14.3	14
	`Prog Lang, User Name`	0.0	10
Learning	`Prog Lang, Library`	53.9	13
	`User Variable, User Func Name`	50.0	10
	`Prog Lang, User Func Name`	46.2	13
	`Prog Lang, Application`	15.4	13
	`Prog Lang, Website`	10.0	10
	`Prog Lang, User Name`	0.0	10
Review	`Device, Library Variable`	40.0	10
	`Version, Data Structure`	37.5	64
	`UI Element, HTML/XML Tag Name`	36.8	38
	`File Name, Website`	10.0	90
	`Application, User Func Name`	9.1	154
	`User Variable, UI Element`	8.9	124
API Change	`Data Structure, Output Block`	75.0	12
	`Algorithm, Error Name`	66.7	12
	`Data Type, UI Element`	63.6	11
	`File Type, Keyboard Input`	17.7	34
	`Library, Website`	17.1	35
	`Application, User Func Name`	17.0	59

while combinations involving the pair (`Programming Language`, `User Name`) results in a 0% success rate, suggesting that providing detailed library information is much more beneficial than including user-specific information.

Regarding the *Discrepancy* intent, we observe a shift towards diagnostic-focused entities such as `Library Function` and `User Variable`. However, combinations involving the pair (`UI Element`, `File Name`) show lower success rates, indicating that backend-related information tends to be more effective for resolution.

Similarly, the *Review* intent tends to occur with entities related to detailed codebase elements, such as `Library Variable`, `Version` and `UI Element`, indicating that these conversations often involve reviewing or resolving problems within specific parts of the codebase.

Our observations show that the SENIR-labelled entities accurately capture the technical content of the questions and reflect the associated intents, as initially intended.

Summary of RQ3

The analysis in RQ3 reveals that specific combinations of software-specific entities and intents impact the likelihood of question resolution. Dominant entities such as `Programming Language` and `Library` play a key role across multiple intents, while entities like `User Variable` and `UI Element` are less beneficial. Success rates vary considerably among intents, with *API Usage* and *API Change* showing the highest resolution rates, while *Discrepancy* and *Review* show lower rates. The Chi-Square analysis confirms that resolution rates significantly differ across intents, suggesting that better question formulation is needed for lower-performing categories.

IV. IMPLICATIONS

In this section, we discuss our findings and their possible implications for developers, chatroom platforms, and researchers.

A. Implications for Developers

It is important to provide focused questions with specific technical details. Our findings (Sections III-B and III-C) reveal that including precise entities, such as `Library Function` or `Library Class`, correlates with higher resolution rates, particularly for intents like *Discrepancy* and *Errors*. Rather than relying on large `CODE SNIPPETS` or abstract references (e.g., `Application`), developers should pinpoint the exact function or class in question.

It is valuable to maintain a positive tone, tag specific users, and avoid overloading with URLs. As highlighted in Section III-B, questions formulated with a positive tone are more likely to be resolved. Likewise, tagging specific users (e.g., "@UserName"), when appropriate, increases visibility and may improve the response rate. However, including too many URLs can overload the conversation and detract from the core question, which can potentially slow down or prevent the resolution.

B. Implications for Chatroom Platforms

Chatroom platforms can benefit from offering structured question templates. Questions that specify concrete entities and intents correlate with higher resolution rates (Section III-C). Chatroom platforms can leverage these insights by providing structured templates that prompt users to include important technical details. For example, if a user indicates they are dealing with an error, the platform could prompt them to specify the `Programming Language`, `Library`, and relevant `Library Function`, to help minimize ambiguity of the questions.

Chatroom platforms can benefit from integrating an approach like SENIR for automated tagging and highlighting. As shown in Section III-A, SENIR effectively labels software-specific entities and intents. Chatroom platforms could integrate SENIR to automatically generate tags for new questions to draw attention to key entities (e.g., `Programming Language`) and identify intents (e.g., *API Usage*). This approach can enable developers to quickly assess a question's context and filter questions based on their expertise.

C. Implications for Researchers

Researchers can use SENIR to label other developer conversation datasets. The results in Section III-A confirm that SENIR can reliably label developer conversations with software-specific entities, intents, and resolution statuses. While our study focuses on Discord, SENIR is generalizable and can be applied to other platforms, such as GitHub Discussions.[4] Researchers can use SENIR to investigate different platforms and analyze how entities and intents influence resolution rates in various contexts.

Researchers can leverage SENIR to enhance chatbots in developer chatrooms. SENIR's entity, intent, and resolution labels provide valuable signals for improving chatbot-based assistance in chatrooms. For example, researchers can use the resolution status to curate high-quality resolved questions for retrieval-augmented generation (RAG) [30]. This enables chatbots to provide more reliable answers based on past conversations. In addition, entities and intents can refine retrieval strategies, techniques such as GraphRAG [20] can build knowledge graphs to enhance retrieval by leveraging labelled entities and intents.

V. Related Work

In this section, we discuss related work about NER in software engineering contexts, LLM applications in software engineering tasks, and developer chatrooms and online forums.

A. NER in Software Engineering

NER has been widely studied in the software engineering domain for automatically identifying and categorizing software-specific entities within text from various sources such as source code, commit messages, documentation, and social media content. Ye et al. [56] developed machine learning based NER systems for software engineering social content. Their approach demonstrates improved performance over rule based systems by addressing entity ambiguity and informal language. Similarly, Tabassum et al. [48] highlighted the importance of domain-specific NER for understanding code-related discussions. Their approach emphasizes the role of software-specific categories like APIs, frameworks, and libraries.

Recent advancements leverage transformer models like BERT for NER tasks, which have improved contextual understanding in software engineering texts. SoftNER [48], a BERT-based model, achieved notable success in identifying code tokens and software-related entities within Stack Overflow conversations. Das et al. [13] further explored zero-shot NER techniques to recognize unseen entities, showcasing the adaptability of pre-trained models in low-resource scenarios. While these studies primarily focus on structured platforms like Stack Overflow, our work targets the fragmented and dynamic nature of developer chatrooms, such as Discord. By leveraging LLMs and integrating intent detection alongside NER, SENIR labels entities and intents to improve question clarity and resolution outcomes in unstructured environments.

[4]https://docs.github.com/en/discussions

B. LLM Applications in Software Engineering

LLMs have shown significant potential in automating a broad range of SE tasks, such as code generation, bug detection, and documentation [16], [23]. Although several studies focus on retrieving Q&A content from structured platforms [57], researchers have also explored LLM-based methods for tasks like requirements engineering [22], [42] and code refinement [18]. Colavito et al. [11] further demonstrated that GPT-like models can effectively classify GitHub issues, reducing human workload in issue labelling.

Our work extends LLM applications to dynamic chatroom environments by introducing automated labelling of entities and intents and leverages the deep contextual understanding of LLMs to enhance question clarity, enabling structured analysis and actionable feedback for developers. This integration extends the applicability of LLMs to dynamic chatroom environments, where conventional retrieval methods struggle.

C. Developer Chatrooms and Online Forums

Developer chatrooms (e.g., Slack, Discord) enable real-time collaboration, but their informal style complicates message analysis. Empirical studies have shown that missing details often lead to unanswered questions, whereas user mentions can boost engagement [14]. Subash et al. [47] introduced the DISCO dataset to highlight the unique challenges of disentangling and analyzing these multi-threaded discussions. Similarly, El Mezouar et al. [15] and Shi et al. [44] found that developer chatrooms contain valuable knowledge but remain difficult to study due to their fluid format.

Lill et al. [32] found that reusing past chats and Q&A posts to resolve new Discord questions is helpful in only 40% of cases—partly because questions often lack clarity. Similarly, Tufano et al. [51] examined how developers use LLMs like ChatGPT to seek assistance in open-source projects, underscoring the growing influence of AI-based aids. While these studies focus on overall chatroom behaviour, our work aims to refine questions by jointly modelling NER, intent, and resolution status within a unified framework. By extracting software-specific entities and understanding the question's underlying purpose, we enable structured insights that pave the way for higher-quality discussions and faster problem resolutions.

VI. Threats to Validity

In this section, we discuss the threats to validity of our study about refining developer questions in chatrooms.

Construct Validity. We study only one LLM, Mixtral, due to its larger token size compared to other open-source models. However, this may limit the generalizability of our results to other models. To mitigate this threat, we carefully select a set of software-specific entities and intent categories that are widely recognized within the software development community. Two PhD students with expertise in software engineering and natural language processing manually label the dataset, ensuring that the entity and intent classification schemes accurately reflect real-world scenarios.

Internal Validity. The primary threat is potential bias in manual labelling and the influence of specific prompt designs on the results. To address this, we employ a rigorous labelling process with two annotators and calculate the inter-annotator reliability using Cohen's Kappa to reduce subjective bias. Moreover, we test different prompt designs in a preliminary study to select the most effective ones for our main experiments.

External Validity. Since our study focuses on Discord chatrooms, the results may not be directly applicable to other software engineering platforms such as Stack Overflow or GitHub Discussions. Furthermore, different communities may have varying norms and communication styles that could affect the performance of our approach. To mitigate this threat, we analyze a large dataset spanning multiple software engineering-related chatrooms to ensure our findings are not specific to a single community. We also select chatrooms with diverse programming topics to improve the applicability of our results. While we acknowledge the diversity in software development communities, we recommend further validation in other environments to ensure broader generalizability.

VII. Conclusion

In this study, we present SENIR, an LLM-based approach for labelling chatroom conversations with software-specific named entities, intents, and resolution outcomes to understand and refine developer questions. Through the lens of three research questions, we demonstrate how these structured labels deepen our understanding of developer Q&A in chatrooms and guide improvements in question formulation. Our experiments on 29,243 conversations from the DISCO dataset showed SENIR's robust performance for entity extraction (**86% F-score**), intent detection (**71% F-score**), and resolution status classification (**89% F-score**). Leveraging these labels, we built predictive models of conversation resolution and found that certain entity-intent combinations (e.g., `Library Function` with *Errors*) increase success, while features like excessive URLs and late posting times hinder resolution. A Chi-Square analysis further confirmed significant differences in resolution rates across various intents, suggesting actionable paths for refining developer questions. Future research can build on our study by exploring real-time feedback mechanisms or extending the approach to additional developer support communities, thereby shaping more targeted, efficient, and high-resolution Q&A.

Acknowledgments

We want to express our appreciation to Bihui Jin at the University of Waterloo for helping to manually validate our results in order to build the golden dataset. His commitment was really helpful in confirming the reliability and precision of our findings.

References

[1] M. Allamanis and C. Sutton, "Why, when, and what: analyzing Stack Overflow questions by topic, type, and code," in *2013 10th Working conference on mining software repositories (MSR)*. IEEE, 2013, pp. 53–56.

[2] A. Anderson, D. Huttenlocher, J. Kleinberg, and J. Leskovec, "Discovering value from community activity on focused question answering sites: a case study of Stack Overflow," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2012, pp. 850–858.

[3] U. Arora, N. Goyal, A. Goel, N. Sachdeva, and P. Kumaraguru, "Ask it right! identifying low-quality questions on community question answering services," in *2022 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2022, pp. 1–8.

[4] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006, pp. 44–54.

[5] S. Beyer, C. Macho, M. Di Penta, and M. Pinzger, "Analyzing the relationships between Android API classes and their references on Stack Overflow," *Univ. Klagenfurt, Klagenfurt, Austria, Tech. Rep. AAU-SERG-2017–002*, 2017.

[6] S. Beyer, C. Macho, M. Pinzger, and M. Di Penta, "Automatically classifying posts into question categories on Stack Overflow," in *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 211–221.

[7] S. Beyer and M. Pinzger, "A manual categorization of Android app development issues on Stack Overflow," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 531–535.

[8] F. Calefato, F. Lanubile, and N. Novielli, "How to ask for technical help? Evidence-based guidelines for writing questions on Stack Overflow," *Information and software technology*, vol. 94, pp. 186–207, 2018.

[9] W.-L. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li, D. Li, B. Zhu, H. Zhang, M. I. Jordan, J. E. Gonzalez, and I. Stoica, "Chatbot arena: an open platform for evaluating llms by human preference," in *Proceedings of the 41st International Conference on Machine Learning*, ser. ICML'24. JMLR.org, 2024.

[10] J. Cohen, *Statistical power analysis for the behavioral sciences*. routledge, 2013.

[11] G. Colavito, F. Lanubile, N. Novielli, and L. Quaranta, "Leveraging GPT-like LLMs to Automate Issue Labeling," in *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 2024, pp. 469–480.

[12] M. Coleman and T. L. Liau, "A computer readability formula designed for machine scoring." *Journal of Applied Psychology*, vol. 60, no. 2, p. 283, 1975.

[13] S. Das, N. Deb, A. Cortesi, and N. Chaki, "Zero-shot Learning for Named Entity Recognition in Software Specification Documents," in *2023 IEEE 31st International Requirements Engineering Conference (RE)*. IEEE, 2023, pp. 100–110.

[14] O. Ehsan, S. Hassan, M. E. Mezouar, and Y. Zou, "An empirical study of developer discussions in the Gitter platform," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 1, pp. 1–39, 2020.

[15] M. El Mezouar, D. A. da Costa, D. M. German, and Y. Zou, "Exploring the use of chatrooms by developers: an empirical study on Slack and Gitter," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3988–4001, 2021.

[16] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 2023, pp. 31–53.

[17] P. Fathollahzadeh, M. El Mezouar, H. Li, Y. Zou, and A. E. Hassan, "Replication Package," https://github.com/Software-Evolution-Analytics-Lab-SEAL/TSE2025-SENIR, 2025.

[18] Q. Guo, J. Cao, X. Xie, S. Liu, X. Li, B. Chen, and X. Peng, "Exploring the potential of ChatGPT in automated code refinement: An empirical study," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[19] E. Guzman and B. Bruegge, "Towards emotional awareness in software development teams," in *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, 2013, pp. 671–674.

[20] H. Han, Y. Wang, H. Shomer, K. Guo, J. Ding, Y. Lei, M. Halappanavar, R. A. Rossi, S. Mukherjee, X. Tang, Q. He, Z. Hua, B. Long, T. Zhao, N. Shah, A. Javari, Y. Xia, and J. Tang, "Retrieval-Augmented Generation with Graphs (GraphRAG)," 2025.

[21] D. J. Hand, "Assessing the performance of classification methods," *International Statistical Review*, vol. 80, no. 3, pp. 400–414, 2012.

[22] S. Hassani, M. Sabetzadeh, and D. Amyot, "An Empirical Study on LLM-based Classification of Requirements-related Provisions in Food-safety Regulations," *arXiv:2501.14683*, 2025.

[23] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, 2024.

[24] Q. Huang, X. Xia, D. Lo, and G. C. Murphy, "Automating Intention Mining," *IEEE Transactions on Software Engineering*, vol. 46, no. 10, pp. 1098–1119, 2020.

[25] Huggingface, "Chat Templates," https://huggingface.co/docs/transformers/main/en/chat_templating, last visited: Feb 23, 2025.

[26] C. Hutto and E. Gilbert, "VADER: A parsimonious rule-based model for sentiment analysis of social media text," in *Proceedings of the international AAAI conference on web and social media*, vol. 8, no. 1, 2014, pp. 216–225.

[27] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand *et al.*, "Mixtral of experts," *arXiv:2401.04088*, 2024.

[28] J. Jiarpakdee, C. Tantithamthavorn, A. Ihara, and K. Matsumoto, "A study of redundant metrics in defect prediction datasets," in *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2016, pp. 51–52.

[29] J. D. M.-W. C. Kenton and L. K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of naacL-HLT*, vol. 1. Minneapolis, Minnesota, 2019, p. 2.

[30] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474.

[31] J. Li, A. Sun, J. Han, and C. Li, "A Survey on Deep Learning for Named Entity Recognition," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 1, pp. 50–70, 2022.

[32] A. Lill, A. N. Meyer, and T. Fritz, "On the helpfulness of answering developer questions on Discord with similar conversations and posts from the past," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[33] M. Liu, X. Peng, Q. Jiang, A. Marcus, J. Yang, and W. Zhao, "Searching StackOverflow questions with multi-faceted categorization," in *Proceedings of the 10th Asia-Pacific Symposium on Internetware*, 2018, pp. 1–10.

[34] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.

[35] M. L. McHugh, "The Chi-square test of independence," *Biochemia medica*, vol. 23, no. 2, pp. 143–149, 2013.

[36] S. Mondal, C. K. Saifullah, A. Bhattacharjee, M. M. Rahman, and C. K. Roy, "Early detection and guidelines to improve unanswered questions on Stack Overflow," in *Proceedings of the 14th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference)*, 2021, pp. 1–11.

[37] T. H. Nguyen, B. Adams, and A. E. Hassan, "Studying the impact of dependency network measures on software quality," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[38] E. Noei, F. Zhang, and Y. Zou, "Too many user-reviews! what should app developers look at first?" *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 367–378, 2019.

[39] S. Noei, H. Li, and Y. Zou, "Detecting Refactoring Commits in Machine Learning Python Projects: A Machine Learning-Based Approach," *arXiv:2404.06572*, 2024.

[40] OpenAI, "Prompt engineering," https://platform.openai.com/docs/guides/prompt-engineering, last visited: Feb 23, 2025.

[41] J. T. Pintas, L. A. Fernandes, and A. C. B. Garcia, "Feature selection methods for text classification: a systematic literature review," *Artificial Intelligence Review*, vol. 54, no. 8, pp. 6149–6200, 2021.

[42] A.-R. Preda, C. Mayr-Dorn, A. Mashkoor, and A. Egyed, "Supporting high-level to low-level requirements coverage reviewing with large language models," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 242–253.

[43] C. Rosen and E. Shihab, "What are mobile developers asking about? a large scale study using Stack Overflow," *Empirical Software Engineering*, vol. 21, pp. 1192–1223, 2016.

[44] L. Shi, X. Chen, Y. Yang, H. Jiang, Z. Jiang, N. Niu, and Q. Wang, "A first look at developers' live chat on Gitter," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 391–403.

[45] I. Srba and M. Bielikova, "Why is Stack Overflow failing? preserving sustainability in community question answering," *Ieee Software*, vol. 33, no. 4, pp. 80–89, 2016.

[46] M.-A. Storey, C. Treude, A. Van Deursen, and L.-T. Cheng, "The impact of social media on software engineering practices and tools," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010, pp. 359–364.

[47] K. M. Subash, L. P. Kumar, S. L. Vadlamani, P. Chatterjee, and O. Baysal, "DISCO: A dataset of Discord chat conversations for software engineering research," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 227–231.

[48] J. Tabassum, M. Maddela, W. Xu, and A. Ritter, "Code and named entity recognition in StackOverflow," *arXiv:2005.01634*, 2020.

[49] C. Treude, O. Barzilay, and M.-A. Storey, "How do programmers ask and answer questions on the web? (NIER track)," in *Proceedings of the 33rd international conference on software engineering*, 2011, pp. 804–807.

[50] J. Tsay, L. Dabbish, and J. Herbsleb, "Let's talk about it: evaluating contributions through discussion in GitHub," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 144–154.

[51] R. Tufano, A. Mastropaolo, F. Pepe, O. Dabić, M. Di Penta, and G. Bavota, "Unveiling ChatGPT's Usage in Open Source Projects: A Mining-based Study," in *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 2024, pp. 571–583.

[52] B. Vasilescu, D. Posnett, B. Ray, M. G. van den Brand, A. Serebrenik, P. Devanbu, and V. Filkov, "Gender and tenure diversity in GitHub teams," in *Proceedings of the 33rd annual ACM conference on human factors in computing systems*, 2015, pp. 3789–3798.

[53] B. Vasilescu, A. Serebrenik, M. Goeminne, and P. Devanbu, "How social Q&A sites are changing knowledge sharing in open source software communities," in *Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing*. ACM, 2014, pp. 342–354.

[54] M. Veera Prathap Reddy, P. Prasad, M. Chikkamath, and S. Mandadi, "NERSE: named entity recognition in software engineering as a service," in *Service Research and Innovation: 7th Australian Symposium, ASSRI 2018, Sydney, NSW, Australia, September 6, 2018, and Wollongong, NSW, Australia, December 14, 2018, Revised Selected Papers 7*. Springer, 2019, pp. 65–80.

[55] X.-L. Yang, D. Lo, X. Xia, Z.-Y. Wan, and J.-L. Sun, "What security questions do developers ask? a large-scale study of Stack Overflow posts," *Journal of Computer Science and Technology*, vol. 31, pp. 910–924, 2016.

[56] D. Ye, Z. Xing, C. Y. Foo, Z. Q. Ang, J. Li, and N. Kapre, "Software-specific named entity recognition in software engineering social content," in *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 90–101.

[57] N. Zhang, Q. Huang, X. Xia, Y. Zou, D. Lo, and Z. Xing, "Chatbot4QR: Interactive query refinement for technical question retrieval," *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1185–1211, 2020.

[58] Z. Zhang, "Variable selection with stepwise and best subset approaches," *Annals of translational medicine*, vol. 4, no. 7, 2016.