Implementation of Hybrid String Matching Approach Using Aho-Corasick for FAQ Chatbot Retrieval System

Arina Azka - 13524049
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: arinaazkaaa@gmail.com , 13524049@std.stei.itb.ac.id

Abstract— **Static FAQ sections require users to browse through long, fixed lists of categories to find a relevant answer, a process that becomes harder as the number of entries grows and does not account for how differently users may phrase the same question. This paper addresses correct-answer retrieval from a curated FAQ knowledge base in response to free-text chatbot queries. To achieve the goal, the problem is modeled using a two-stage hybrid string matching pipeline that combines lexical keyword matching with semantic reranking. In this model, queries are normalized into stemmed keyword tokens, with each FAQ entry assigned a relevance score indicating the strength of its match to the query, while each matched candidate is assigned a value representing its BM25 ranking weight. To find a solution to the retrieval problem, this paper utilizes the Aho-Corasick algorithm for simultaneous multi-keyword matching, complemented by Levenshtein distance for typo-tolerant fallback matching and Sentence-BERT embeddings for semantic reranking when lexical matching is insufficient. The implementation is carried out using a Python program integrated with the LINE Messaging API that is verified through five scenarios: exact keyword matching, typographical errors, synonym substitution, semantic paraphrasing, and referential follow-up queries. The results of the tests indicate that the pipeline correctly retrieves answers for the query variations covered in each scenario. In conclusion, the FAQ retrieval problem was addressed by a hybrid pipeline of Aho-Corasick, Levenshtein distance, BM25, and sentence embeddings, which offers tolerant and accurate query matching for an FAQ chatbot.**

Keywords—string matching; Aho-Corasick; Levenshtein distance; BM25; FAQ chatbot

I. Introduction

Browsing through a long list of FAQ sections to find one relevant answer is rarely how users actually want to get help. Most users would rather type or say what they need directly and have an answer come to them, instead of scrolling, scanning headers, and guessing which section their question might fall under. This preference for direct interaction is reinforced by two practical shortcomings of static FAQ sections. First, as an FAQ page accumulates more entries over time, the list of sections users must sift through grows accordingly, working against the very purpose of self-service that FAQ pages are meant to serve. Second, FAQ sections are organized under fixed category titles chosen by the page's author, which means a user must still mentally match their own phrasing to whichever heading the author happened to choose, particularly difficult when a question spans more than one category, such as a cancellation request that also involves a refund. In effect, the user is left to perform the same matching task a system could otherwise automate.

This is exactly what motivates the use of chatbots as a customer service interface, since a chatbot lets a user simply ask in their own words and receive an answer immediately, without first locating the right page or category. However, not all chatbot designs deliver on this promise of convenience equally well. Menu or button-based chatbots, which guide users through a fixed sequence of predefined choices, sidestep the ambiguity of free text but at the cost of flexibility: a user is confined to whatever options the designer anticipated, and any question that does not fit neatly into the provided menu structure simply has no path to an answer. Free-text chatbots avoid this rigidity by allowing users to type naturally, but many of them are themselves too rigid in a different way. Rule-based chatbots answer queries by looking for pattern matches, and are consequently likely to produce inaccurate answers whenever a query does not contain any of the known patterns. A user typing "how long does delivery take" receives no answer because the stored entry says "shipping time," and a user who mistypes a single letter in an otherwise correct keyword is met with the same fallback message as someone asking something entirely irrelevant. The user is then left to guess at alternate phrasings, retype their question, or give up and wait for a human agent, defeating the purpose of having an automated FAQ system in the first place.

The deeper issue is not that these systems are poorly implemented, but that manually encoding pattern matching rules is difficult and time consuming, and the resulting rules tend to be brittle and domain specific, transferring poorly from one problem to another. An FAQ author cannot realistically anticipate every way a question might be asked, and even a single missing keyword variant or a single typo is enough for an otherwise correctly designed system to fail a user it should have been able to help.

Closing this gap does not require abandoning string matching in favor of a full machine learning pipeline. It only requires making the matching itself more tolerant of surface-level variation, a misspelling, a missing word, a different word order. The Aho-Corasick algorithm constructs a single finite-state automaton from a set of keywords and scans candidate text in a single pass, achieving time complexity linear in the combined length of all patterns, the length of the text, and the number of matches found, making it well suited for matching multiple query keywords against a knowledge base of FAQ entries simultaneously. To complement exact pattern matching with tolerance for typographical error, Levenshtein distance measures the minimum number of single-character insertion, deletion, and substitution operations required to transform one string into another, enabling approximate retrieval when an exact match is not found. Candidates identified through these matching techniques are then ranked using BM25, a term-weighting scheme that scores candidates based on keyword frequency and rarity across the FAQ collection.

This paper implements the proposed FAQ chatbot on LINE, a messaging platform with wide everyday adoption, so that users can ask questions through a chat interface they already use rather than navigating to a separate website or search bar. Based on these considerations, this paper proposes an FAQ chatbot that answers user queries against a curated FAQ knowledge base using Aho-Corasick multi-keyword matching as its primary retrieval mechanism, Levenshtein distance-based approximate matching as a fallback for typographically imprecise queries, and BM25 for ranking matched candidates. The effectiveness of the system is evaluated using precision, recall, and F1-score metrics.

II. Basic Theory

A. String Matching

String matching, or pattern matching, is the process of locating occurrences of a pattern string P of length m within a longer text string T of length n, where m is assumed to be significantly smaller than n. This technique is foundational to a wide range of computing applications including text editors, search engines, bioinformatics, and information retrieval systems. In general, string matching approaches can be categorized into two types. Exact matching seeks to find positions in the text where the pattern appears character-for-character with no deviation. Fuzzy matching, on the other hand, allows for approximate matches by tolerating typographical errors, spelling variations, and other forms of minor discrepancy between the query and the target text.

B. Aho-Corasick Algorithm

The Aho-Corasick algorithm is a multi-pattern string matching algorithm that enables the simultaneous search of multiple pattern strings within a text. Aho-Corasick constructs a single finite state automaton from the entire set of patterns, known as a dictionary, and processes the text only once regardless of the number of patterns being searched. Given a dictionary with a total pattern length of m and an alphabet of size k, the algorithm constructs the automaton in O(mk) time and subsequently processes a text of length n in O(n + output) time, where output refers to the total number of matches found. The algorithm was first proposed by Alfred Aho and Margaret Corasick in 1975.

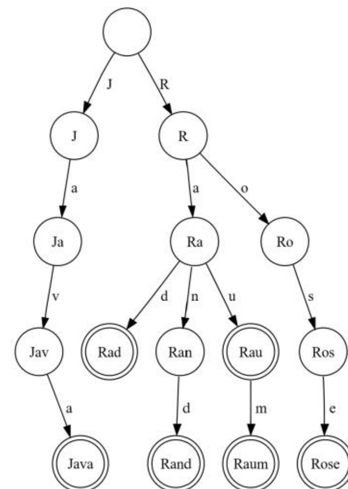

Figure 1. Trie Construction

The foundation of the Aho-Corasick algorithm is a trie, a rooted tree structure in which each edge is labeled with a character and no two outgoing edges from the same vertex share the same label. Each vertex in the trie corresponds to a string formed by the sequence of edge labels along the path from the root to that vertex. Vertices that correspond to complete pattern strings in the dictionary are marked with an output flag. The trie is constructed by inserting each pattern string character by character, starting from the root. If a required edge does not yet exist, a new vertex is created. Construction of the trie runs in linear time with respect to the total length of all patterns and requires O(mk) memory, where m is the total pattern length and k is the alphabet size.

Once the trie is built, the Aho-Corasick algorithm extends it into a finite deterministic automaton by adding suffix links to every vertex. A suffix link for a vertex v points to the vertex corresponding to the longest proper suffix of v's string that also exists as a prefix of some pattern in the trie. The root vertex's suffix link points to itself. For all children of the root, their suffix links also point to the root. For deeper vertices, the suffix link is computed recursively: given vertex v with parent p reached by edge character c, the suffix link of v is found by following the suffix link of p and then taking the transition for character c from there.

These suffix links serve a critical purpose during text processing. When the automaton is in a state corresponding to string t and needs to transition using character c, but no outgoing edge labeled c exists, it follows the suffix link and retries the transition from there. This process repeats until a valid transition is found or the root is reached. This mechanism ensures that the automaton always maintains the longest partial match found so far, without ever moving backwards through the input text.

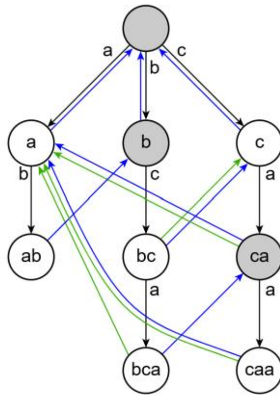

Figure 2. Aho-Corasick Automaton

Once the automaton is fully constructed, the text is processed letter by letter from left to right. At each position, the automaton transitions to the next state according to the current character. A match is detected not only when the current state is an output vertex, but also when any output vertex is reachable by following the chain of suffix links from the current state. To avoid the O(n * len) overhead of traversing suffix links at every position, each vertex can precompute an exit link, which points directly to the nearest output vertex reachable via suffix links. This optimization reduces match detection to O(1) per position, achieving an overall text processing complexity of O(n + ans), where ans is the total number of matches.

C. Levenshtein Distance

Levenshtein distance is a metric for measuring the similarity between two strings by quantifying the minimum number of single-character edit operations required to transform one string into the other. The three permitted operations are insertion, which adds a character to a string, deletion, which removes a character from a string, and substitution, which replaces one character with another. The concept was introduced by Vladimir Levenshtein in 1965 and remains one of the most widely used string similarity metrics in computing.

As a simple illustration, transforming the string "kitten" into "sitting" requires three operations, substituting "k" with "s", substituting "e" with "i", and inserting "g" at the end. The Levenshtein distance between the two strings is therefore 3. In general, the distance is always non-negative and equals zero if and only if the two strings are identical. The maximum possible distance between two strings of lengths m and n is max(m, n), occurring when one string must be completely replaced by the other through insertions or deletions alone.

		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

Figure 3. DP Matrix for Levenshtein Distance

The Levenshtein distance between two strings a and b can be computed efficiently using dynamic programming. A matrix of size (m+1) × (n+1) is constructed, where m and n are the lengths of strings a and b respectively. The first row is initialized with values 0, 1, 2, ..., n representing the cost of deleting all characters of b, and the first column is initialized with values 0, 1, 2, ..., m representing the cost of deleting all characters of a. Each subsequent cell dp[i][j] is filled according to the following rule.

$$dp[i][j] = dp[i-1][j-1], if\ a[i] = b[j]$$

$$dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]), otherwise$$

The three candidates in the second case correspond to the costs of deletion (dp[i-1][j]), insertion (dp[i][j-1]), and substitution (dp[i-1][j-1]) respectively. The final Levenshtein distance is found at the bottom-right cell dp[m][n] of the completed matrix. This approach runs in O(m×n) time and O(m×n) space, and can be further optimized to O(min(m,n)) space by storing only two rows of the matrix at a time.

D. Term Weighting and BM25

Term weighting is a technique used to assign numerical importance to each term in a document or query, so that terms can be compared and ranked rather than simply checked for presence or absence. The most foundational term weighting scheme is term frequency-inverse document frequency (TF-IDF), which combines two opposing signals: term frequency (TF), how often a term appears within a single document, and inverse document frequency (IDF), how rare that term is across the entire document collection. A term that appears frequently in one document but rarely across the collection receives a high weight, since it is likely to be discriminative for that document, while a term that appears in nearly every document receives a low weight, since its presence carries little information about relevance. BM25 (Best Matching 25) extends this idea into a complete probabilistic scoring function for ranking documents against a query. Given a query Q containing terms, the BM25 score of a document D is computed as the sum over all query terms of each term's IDF weight multiplied by a saturated term frequency component.

$$Score(D,Q) = \sum_{i=1}^{n} IDF(q_i) \cdot \frac{f(q_i,D) \cdot (k_1+1)}{f(q_i,D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{avgDL}\right)}$$

Here, f(qi, D) is the frequency of query term qi in document D, |D| is the length of document D, and avgdl is the average document length across the collection. The parameter k1 controls how quickly additional occurrences of a term stop contributing to the score, preventing a document from being ranked disproportionately higher simply because a term appears many times, while the parameter b controls the strength of length normalization, penalizing longer documents that may contain query terms only by chance due to their size. Unlike raw TF-IDF, this saturation behavior means that going from one occurrence of a term to two contributes a smaller score increase than going from zero to one, more closely reflecting how additional repetitions provide diminishing evidence of relevance. BM25 has become a standard baseline ranking function in information retrieval systems due to its strong empirical performance and its independence from any training data, since the formula is entirely derived from term and document statistics already present in the collection.

E. Sentence Embedding and Semantic Similarity

A sentence embedding is a fixed-length numerical vector representation of a sentence, constructed such that sentences with similar meaning are positioned close to one another in the resulting vector space, even when they share few or no words in common. This stands in contrast to lexical representations such as the term vectors used in TF-IDF or BM25, which can only register a relationship between two sentences if they share overlapping vocabulary. Sentence-BERT (SBERT) is a widely used method for producing such embeddings, built by fine-tuning a pretrained BERT-based language model using a siamese network structure, in which two copies of the same network process a pair of sentences independently and are trained so that the resulting embeddings of semantically similar sentence pairs end up close together in vector space. Once trained, SBERT can encode a single sentence into an embedding vector through one forward pass through the network, after which the embedding can be compared against other precomputed embeddings using lightweight vector operations rather than requiring the language model itself to process every sentence pair directly, substantially reducing the computational cost of comparing a query against a large collection of candidate sentences. The similarity between two sentence embeddings is most commonly measured using cosine similarity, which quantifies how closely two vectors point in the same direction regardless of their magnitude. Given two embedding vectors u and v, cosine similarity is computed.

$$cos(u,v) = (u \cdot v) / (||u|| \cdot ||v||)$$

The result ranges from -1 to 1, where a value close to 1 indicates that the two sentences are semantically similar, a value close to 0 indicates no particular relationship, and a value close to -1 indicates that the sentences are semantically opposed. When embedding vectors are normalized to unit length beforehand, as is standard practice for sentence embedding models, the cosine similarity calculation reduces to a simple dot product between the two vectors, since the denominator term ||u|| · ||v|| becomes equal to 1.

III. IMPLEMENTATION

The chatbot is implemented as a Python backend integrated with the LINE Messaging API through a webhook, using Flask to handle incoming requests. Text matching and ranking are implemented purely with the algorithms from Section II, supported by Sastrawi for Indonesian stemming and sentence-transformers for semantic reranking. The FAQ knowledge base is stored as a JSON file and processed into a serialized index using the pickle module, so it does not need to be rebuilt on every server restart.

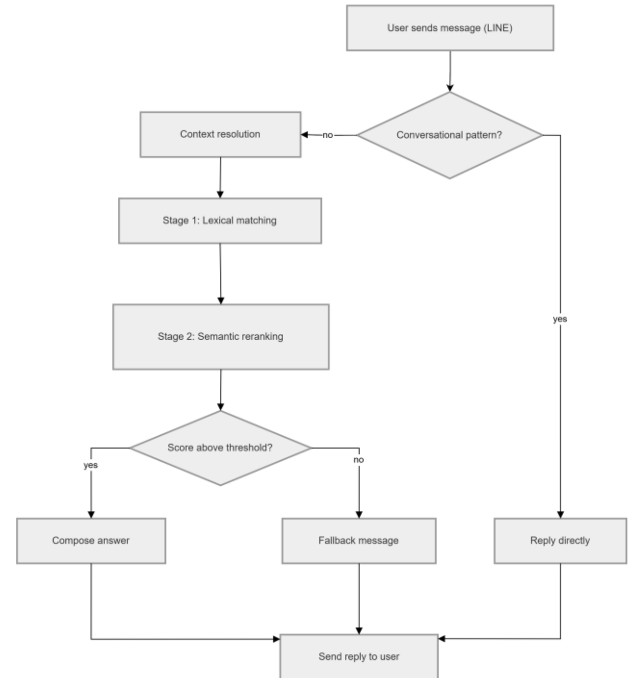

Figure 4. Message Processing Flow of the FAQ Chatbot

A message is first checked against simple conversational patterns (greetings, thanks), which get a direct reply. Otherwise, context resolution completes referential or elliptical queries using the entity from the previous turn, before passing through Stage 1 lexical matching (Aho-Corasick, Levenshtein, BM25) and Stage 2 semantic reranking (sentence embeddings). If the best candidate's score exceeds the threshold, an answer is composed. Otherwise, a fallback message is returned. Both paths converge in the final reply sent to the user.

A. Preprocessing and Index Construction

Each stored FAQ question is normalized through a four-step pipeline in indexing.py, namely removal of non-alphanumeric characters via regular expression, tokenization with stopword removal, and stemming using Sastrawi, which reduces inflected words such as "pengiriman" to their root form "kirim" so that different word forms can still be matched.

```
def tokenize(text):
```

```
    text = text.lower()
    text = re.sub(r"[^a-z0-9\s]", " ", text)
    return [t for t in text.split() if t and t not in STOPWORDS]


def stem_tokens(tokens):
    return [stem_word(t) for t in tokens]


def analyze(text):
    return stem_tokens(tokenize(text))
```

Stemming results are cached with functools.lru_cache since the same word recurs often across FAQ entries. Each processed entry is stored as an index record containing its stemmed tokens and document length, while an inverted index mapping each term to the sentence IDs containing it is built in the same pass, later used to restrict which candidates need scoring during search.

B. Conversational Context Resolution

Since follow-up questions often omit their subject or refer back to it with a pronoun, context.py tracks the last entity mentioned by each user. A query is treated as referential if it contains a pronoun-like word such as "itu" or the "-nya" suffix, or as elliptical if no content word remains after stripping question prefixes. In either case, the stored entity is appended to the query before it proceeds to search, which works because matching downstream is keyword-based via Aho-Corasick and unaffected by word order.

C. Stage 1: Lexical Matching and BM25 Ranking

The raw query is first converted into search terms using the prepare_query_terms function, where question words are removed, the remaining tokens are stemmed, and any term not found in the vocabulary is corrected using Levenshtein distance through the closest_vocabulary_term function. The term list is then expanded with synonym groups (synonyms.py) so that, for example, "harga" and "biaya" resolve to the same set.

```
matcher = AhoCorasick(terms)
candidate_ids = set()
for term in terms: candidate_ids |= index["postings"].get(term, set())
```

The inverted index limits scoring to only those FAQ entries sharing at least one term with the query. For each candidate, the Aho-Corasick automaton reports matching terms and their frequencies, which are passed into the BM25 scoring function to produce a relevance score, optionally boosted when the detected query intent (price, shipping, refund) matches a monetary or date pattern in the candidate answer.

D. Stage 2: Semantic Reranking

Lexical matching alone cannot capture cases where a query and an FAQ entry express the same intent with different vocabulary. The top-ranked Stage 1 candidates are therefore reranked using sentence embeddings from a pretrained multilingual sentence-transformer model (embedding.py), encoded once per FAQ entry at indexing time and once per query at request time.

```
cos = float(embeddings[sentence["id"]] @ qv)
final = cos + LEX_NUDGE * bm_norm
```

Because embeddings are normalized to unit length, cosine similarity reduces to a dot product. The final score blends this semantic similarity with a small weighted contribution from the normalized BM25 score, and any candidate below a minimum similarity threshold is discarded to avoid returning an unrelated answer with false confidence.

E. LINE Webhook Integration

The Flask application exposes a /callback endpoint that verifies incoming requests using the LINE channel secret before dispatching them to the message handler. Simple conversational patterns, such as greetings, are answered directly. Otherwise, the message is resolved against the user's stored context, passed through the two-stage retrieval pipeline, and the resulting answer is returned as a reply. User context is kept in an in-memory dictionary keyed by LINE user ID, sufficient for a single-process deployment.

IV. TESTING

Testing covered five scenarios, each isolating one linguistic challenge in the retrieval pipeline, namely exact keyword matching, typographical errors, synonym substitution, semantic paraphrasing, and referential follow-up queries.

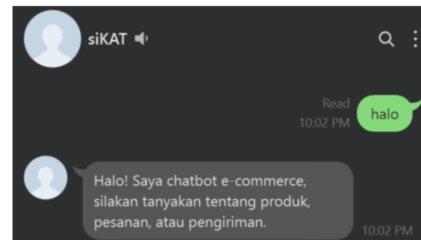

Figure 5. Example of a Direct Reply to a Conversational Greeting Pattern

A. Scenario 1: Exact Keyword Matching

This scenario tests whether a query whose stemmed keywords directly match a stored FAQ entry is retrieved correctly to establish the baseline performance of Stage 1 alone.

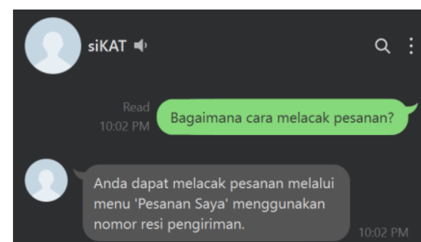

Figure 6. Chatbot Response to an Exact Keyword Match Query

The query "Bagaimana cara melacak pesanan?" was answered correctly, returning the FAQ entry on order tracking via the 'Pesanan Saya' menu using a shipment receipt number. This confirms that Aho-Corasick successfully detects stemmed keywords such as "lacak" and "pesan" in a single pass.

B. *Scenario 2: Typographical Errors*

This scenario tests whether a single-letter typo in the main keyword is corrected to the right vocabulary term using Levenshtein distance.

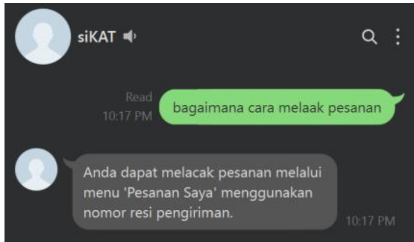

Figure 7. Chatbot Response to a Typographical Errors Query

The query "bagaimana cara melaak pesanan" (a missing letter in "melacak") was answered correctly, returning the tracking entry. This confirms that typo correction works for at least this typo pattern, though an earlier test with a different typo ("melakcak") on the same word failed, suggesting the correction is not yet reliable across all typo types.

C. *Scenario 3: Synonym Substitution*

This scenario tests whether a query using a synonym instead of the word stored in the FAQ entry still retrieves the correct answer.

Figure 8. Chatbot Response to a Synonym Substitution

The query "berapa price kirim barang" was answered correctly, returning the FAQ entry on shipping cost calculation. This confirms that the synonym expansion mechanism correctly maps "price" to "biaya," the term actually stored in the FAQ entry, allowing the system to retrieve the correct answer even though the query does not use the FAQ database's original vocabulary.

D. *Scenario 4: Semantic Paraphrasing*

This scenario tests whether a query with little to no lexical overlap with the target entry is still retrieved through semantic reranking.

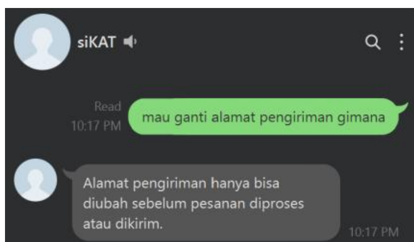

Figure 9. Chatbot Response to a Semantic Paraphrasing Query

The query "mau ganti alamat pengiriman gimana" was answered correctly, returning the entry on changing shipping addresses despite sharing almost no stemmed vocabulary with it. This confirms the embedding model can capture intent beyond surface wording, though an earlier, more divergent paraphrase ("barangnya nggak sampe") failed, indicating this capability is not yet consistent across all paraphrase types.

E. *Scenario 5: Referential Follow-up*

This scenario tests whether an elliptical follow-up query is correctly completed using the entity resolved from the previous conversational turn.

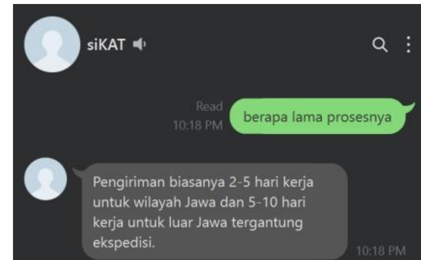

Figure 10. Chatbot Response to a Referential Follow-up Query

The follow-up "berapa lama prosesnya" was correctly resolved using the shipping-related entity from the prior turn, returning the correct delivery time entry. This confirms context resolution works, though since the prior topic was already shipping-related, this case does not yet rule out failure when the carried-over entity is ambiguous or unrelated to the current query.

The testing results show that the chatbot can correctly retrieve FAQ answers across different query variations. Exact keyword queries are handled well using lexical matching and BM25, while synonym substitution works when mapped terms exist in the synonym dictionary. The system can also handle minor typographical errors through Levenshtein-based correction and capture some semantic and referential variations using sentence embeddings and context resolution. However, performance degrades for more complex typos and distant paraphrases, indicating dependence on vocabulary coverage, synonym quality, and similarity thresholds. Overall, the hybrid approach improves retrieval robustness compared to exact matching, but still has limitations when queries deviate significantly from the FAQ data.

V. Conclusion

This paper presents the implementation of a hybrid string matching approach for an FAQ chatbot retrieval system integrated with the LINE Messaging API. The system combines several retrieval techniques, including Aho-Corasick for multi-keyword matching, Levenshtein distance for typo correction, BM25 for lexical ranking, synonym expansion for vocabulary variation, and sentence embeddings for semantic reranking. These components allow the chatbot to retrieve answers from a curated FAQ knowledge base without relying entirely on generative language models.

Based on the testing results, the proposed system successfully handles exact keyword queries, minor typographical errors, synonym substitution, semantic paraphrasing, and simple referential follow-up queries. The results indicate that combining lexical and semantic retrieval improves the chatbot's ability to answer user questions even when the input does not exactly match the stored FAQ text. Aho-Corasick and BM25 provide efficient and reliable retrieval when the query shares vocabulary with the FAQ

database, while Levenshtein correction and synonym expansion help reduce failures caused by surface-level variations. Sentence embedding reranking further improves retrieval when the user's wording differs from the stored FAQ entry.

However, the system still has several limitations. Typo correction is only reliable for errors within a small edit distance, synonym handling depends on manually defined synonym groups, and semantic reranking may fail when the paraphrase is too far from the original FAQ wording. In addition, context resolution only works for relatively simple follow-up queries and may produce incorrect results when the previous entity is ambiguous. Future improvements may include expanding the FAQ dataset, enriching the synonym dictionary, improving typo correction with phonetic or language-specific rules, and evaluating the system using a larger set of queries with quantitative metrics such as precision, recall, and F1-score.

APPENDIX

Source code used in Section III Implementation can be viewed here:

https://github.com/arinazk/FAQ-chatbot

ACKNOWLEDGMENT

The author deeply conveys heartfelt gratitude to Allah SWT, who has bestowed grace and blessings to complete this paper, titled “Implementation of Hybrid String Matching Approach Using Aho-Corasick for FAQ Chatbot Retrieval System”. The author would also extend appreciation to Dr. Rinaldi, a lecturer in IF2211 Algorithm Strategy, for the dedication, guidance, encouragement, and motivation throughout the semester. The author is genuinely thankful for the endless support and strength coming from the author's family and friends.

REFERENCES

[1] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, Jun. 1975.

[2] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[3] S. Robertson and H. Zaragoza, "The probabilistic relevance framework: BM25 and beyond," *Foundations and Trends in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[4] G. Caldarini, S. Jaf, and K. McGarry, "A literature survey of recent advances in chatbots," *Information*, vol. 13, no. 1, p. 41, Jan. 2022.

[5] R. Munir, "Pencocokan String (String/Pattern Matching)," Bahan Kuliah IF2211 Strategi Algoritma, Program Studi Teknik Informatika STEI-ITB, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/23-Pencocokan-string-(2026).pdf. [Accessed June 14, 2026]

[6] cp-algorithms, "Aho-Corasick algorithm," Algorithms for Competitive Programming, Apr. 2025. [Online]. Available: https://cp-algorithms.com/string/aho_corasick.html. [Accessed June 14, 2026]

[7] GeeksforGeeks, "Introduction to Levenshtein Distance," GeeksforGeeks, Jan. 2024. [Online]. Available: https://www.geeksforgeeks.org/dsa/introduction-to-levenshtein-distance/. [Accessed June 14, 2026]

[8] E. Nam, "Understanding the Levenshtein Distance Equation for Beginners," Medium, Feb. 2019. [Online]. Available: https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0. [Accessed June 14, 2026]

[9] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using Siamese BERT-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Hong Kong, China, Nov. 2019, pp. 3982–3992.

STATEMENT

I hereby declare that this paper is my original work, not an adaptation or translation of any other individual’s work, and not plagiarism.

Bandung, June 19, 2026

Arina Azka - 13524049