A Comprehensive Study on Fact-checking Social Media Content using RAG and LLM

by

Md Abid Hossain
ID: CSE2203027033

Md Junaed Tanvir
ID: CSE2203027064

Md Miraj
ID: CSE2203027023

Haymonti Rajbongshi
ID: CSE2203027066

Supervised by
Mohammad Naderuzzaman

Submitted in partial fulfillment of the requirements for the degree of
Bachelor of Science in Computer Science and Engineering



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SONARGAON UNIVERSITY (SU)

September 2026

APPROVAL

The thesis titled "**A Comprehensive Study on Fact-checking Social Media Content using RAG and LLM**" submitted by Md. Abid Hossain (CSE2203027033), Md Junaed Tanvir (CSE2203027064), Md Miraj (CSE2203027023) and Haymonti Rajbongshi (CSE2203027066) to the Department of Computer Science and Engineering, Sonargaon University (SU), has been accepted as satisfactory for the partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science and Engineering and approved as to its style and contents.

Board of Examiners

Mohammad Naderuzzaman
Associate Professor,
Department of Computer Science and Engineering
Sonargaon University (SU)

Supervisor

(Examiner Name and Signature)
Department of Computer Science and Engineering
Sonargaon University (SU)

Examiner 1

(Examiner Name and Signature)
Department of Computer Science and Engineering
Sonargaon University (SU)

Examiner 2

(Examiner Name and Signature)
Department of Computer Science and Engineering
Sonargaon University (SU)

Examiner 3

DECLARATION

We, hereby, declare that the work presented in this report is the outcome of the investigation performed by us under the supervision of **Mohammad Naderuzzaman, Associate Professor,** Department of Computer Science and Engineering, Sonargaon University, Dhaka, Bangladesh. We reaffirm that no part of this thesis has been or is being submitted elsewhere for the award of any degree or diploma.

Countersigned

(Mohammad Naderuzzaman)
Supervisor

Signature

Md Abid Hossain
ID: CSE2203027033

Md Junaed Tanvir
ID: CSE2203027064

Md Miraj
ID: CSE2203027023

Haymonti Rajbongshi
ID: CSE2203027066

ABSTRACT

The fast development of digital media and online news websites has made it easier and quicker for people to get information. However, the spread of incorrect, misleading, and possibly false information has become a major problem for individuals looking for accurate and trustworthy sources. In response to this situation, this research introduces a dynamic agentic fact-checking system. This system combines Natural Language Processing (NLP), Deep Learning, Large Language Models (LLMs), and Retrieval-Augmented Generation (RAG) to verify news claims in real time. The system uses the "openai/gpt-oss-120b", "openai/gpt-oss-20b", "qwen/qwen3.6-27b" Large Language Model via the Groq API to analyze user input in Bangla, English, and Banglish. The process begins with NLP to detect important words and names. Then, a web search using DuckDuckGo fetches up-to-date and relevant information from online news sources. The system analyzes the collected information alongside the original claim to determine whether the claim is accurate. Finally, it provides a well-structured, impartial verdict along with a detailed explanation based on the evidence found. Unlike traditional fact-checking methods that rely on fixed datasets or pre-existing knowledge, this system retrieves current online evidence before forming a conclusion. This design allows the system to check information in real time and explain its findings with supporting data. The proposed system shows how combining NLP, LLMs, Agentic AI, and RAG can create an effective and flexible fact-checking tool for verifying information across multiple languages.

Keywords: Fact-Checking, Agentic AI, Retrieval-Augmented Generation (RAG), Large Language Models (LLMs), Natural Language Processing (NLP), Fake News Detection, Real-Time Information Verification, Multilingual Analysis.

ACKNOWLEDGMENT

At the very beginning, we would like to express our deepest gratitude to the Almighty Allah for giving us the ability and the strength to finish the task successfully within the scheduled time.

We are auspicious that we had the kind association as well as supervision of **Mohammad Naderuzzaman, Associate Professor**, Department of Computer Science and Engineering, Sonargaon University whose hearted and valuable support with the best concern and direction acted as necessary recourse to carry out our project.

We would like to convey our special gratitude to **Brig. Gen. (Retd) Prof. Habibur Rahman Kamal, ndc, psc** Dean, Department of Computer Science and Engineering and **Prof. Bulbul Ahamed**, Head, Department of Computer Science and Engineering, for his kind concern and precious suggestions.

We are also thankful to all our teachers during our whole education for exposing us to the beauty of learning.

Finally, our deepest gratitude and love to our parents for their support, encouragement, and endless love

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
CMS	Content Management System
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
GPU	Graphics Processing Unit
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JS	JavaScript
JSON	JavaScript Object Notation
LLM	Large Language Model
NLP	Natural Language Processing
ORM	Object-Relational Mapping
RAG	Retrieval-Augmented Generation
SDLC	Software Development Life Cycle
UI	User Interface

TABLE OF CONTENTS

Title	Page No.
APPROVAL	ii
DECLARATION	iii
ABSTRACT	iv
ACKNOWLEDGMENT	v
LIST OF ABBREVIATIONS	vi
LIST OF FIGURES	ix
CHAPTER 1 - LIVE AI-BASED FACT VERIFICATION FOR BENGALI NEWS CLAIMS	1
1.1 Introduction	1
1.2 Project Motivation	3
1.3 Problem Statement	5
1.4 Objectives	6
1.5 Challenges	7
CHAPTER 2 - SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC)	9
2.1 Introduction	9
2.2 Software Development Life Cycle Models	10
2.3 Iterative Model Design	10
2.4 Why the Iterative Model is Better for LiveFactChecker	12
CHAPTER 3 - DEVELOPMENT TOOLS AND TECHNOLOGIES	14
3.1 Introduction	14
3.2 Programming Language and Core Libraries	15
3.3 Front-End Framework	16
3.4 Dependency Management	17
3.5 Development Environment and Configuration	17
3.6 System and Browser Requirements	17
CHAPTER 4 - SYSTEM ARCHITECTURE AND DESIGN	18
4.1 Introduction	18

4.2 Data Objects and Flow 18
4.3 Use Case Diagram 20
4.4 Data-Flow Diagram 21
CHAPTER 5 - LIVEFACTCHECKER: IMPLEMENTATION 23
5.1 Introduction 23
5.2 Back-End Implementation (fact_checker.py) 23
5.2.1 Initialization 23
5.2.2 Multi-Angle Query Generation 23
5.2.3 Evidence Retrieval 24
5.2.4 Verdict Generation 25
5.3 Front-End Implementation (app.py) 26
5.3.1 Page Configuration and Styling 26
5.3.2 Session Setup and Input 26
5.3.3 Handling the Check and Displaying Results 27
5.4 End-to-End Walkthrough 28
CHAPTER 6 - LIMITATIONS AND FUTURE WORKS 34
6.1 Limitations 34
6.2 Future Works 35
CHAPTER 7 - CONCLUSION 37
REFERENCES 38

LIST OF FIGURES

Figure No.	Title	Page No.
Fig. 2.1	SDLC	9
Fig. 2.2	Iterative Model	12
Fig. 3.1	LiveFactChecker Technology Stack	14
Fig. 4.1	Data Objects and Flow through LiveFactChecker	19
Fig. 4.2	Use Case Diagram	20
Fig. 4.3	Data-Flow Diagram (Level 0 - Context Diagram)	22
Fig. 5.1	Application Home Page	28
Fig. 5.2	Verdict - Real / Reported Example	29
Fig. 5.3	Verdict - Fake / Misleading Example	30
Fig. 5.4	Verdict - Developing / Unverified Example	31
Fig. 5.5	Evidence Sources - Real Claim	32
Fig. 5.6	Evidence Sources - Fake Claim	33

CHAPTER 1

LIVE AI-BASED FACT VERIFICATION FOR BENGALI NEWS CLAIMS

1.1 Introduction

Misinformation is not a new phenomenon, but the environment in which it now spreads is. Before the era of social media, a false or distorted claim travelled only as fast as word of mouth, print circulation, or broadcast schedules allowed, giving journalists and institutions time to intervene before it reached a wide audience. That constraint has effectively disappeared. Platforms such as Facebook, WhatsApp, and YouTube allow a single post, screenshot, or short video to reach an audience of thousands or millions within hours, often accompanied by no more evidence than the claim's own confident tone.[5], [6] predict whether the evidence supports, refutes, or is insufficient to judge the claim This shift has turned misinformation from a manageable editorial problem into a large-scale information-ecosystem problem, and it is the broader field within which this project is situated: the use of computational methods - specifically natural language processing (NLP) and large language models (LLMs) - to make claim verification fast enough to keep pace with how claims actually spread.

Automated fact-checking as a research field grew out of two separate traditions that have increasingly converged. The first is journalistic fact-checking, historically a manual craft practised by dedicated organisations that investigate a claim, gather documentary or expert evidence, and publish a verdict - thorough, but slow relative to the speed of online sharing. The second is the NLP research tradition of claim detection and verification, which treats fact-checking as a text-classification or natural-language-inference problem: given a claim and a body of evidence, predict whether the evidence supports, refutes, or is insufficient to judge the claim. [2],[3] Early systems in this tradition, and the datasets built to train them, generally worked against a fixed, pre-collected body of evidence[2], which meant they inherited a version of the same weakness a static LLM has: they could reason well about claims tied to the evidence they were given, but had no mechanism for reaching outside that fixed evidence to check something that happened after the fact.

Large language models have reshaped this landscape twice over. On one hand, models trained on very large text corpora are able to read a claim, recognise its subject and named

entities, and produce fluent, well-structured reasoning about it, which makes them attractive as a component in an automated fact-checking pipeline. [15] On the other hand, an LLM used in isolation is fundamentally limited by what it saw during training. Its knowledge has a cut-off date, it cannot observe events that occurred afterward, and - perhaps more damaging in a fact-checking context - it can produce a fluent, confident-sounding answer even when it is wrong, a behaviour widely referred to as hallucination. [9], [14], [20] Retrieval-augmented generation (RAG) is the approach that has emerged in the NLP field to address this specific weakness: rather than asking a model to answer purely from what it learned during training, the model is first given evidence retrieved from an external, up-to-date source, and is asked to reason from that evidence.[1] This project situates itself directly within that line of work, applying a retrieval-augmented approach to the specific and comparatively underserved problem of Bengali-language fact-checking.

Most existing automated fact-checking tools, and the great majority of the datasets and research built to support them, are designed for English[2], [3], [4]. Bengali, despite being spoken by more than two hundred million people and despite Bangladesh having one of the highest social-media-driven news consumption rates in South Asia, has very little dedicated fact-checking tooling, and what does exist is largely manual. This gap is compounded by a further linguistic complication that is specific to how Bengali speakers actually write online: a large share of everyday communication happens not in Bangla script but in Banglish, a Romanized, code-mixed form of Bengali. A claim verification tool that only understands one script or one language therefore excludes a substantial share of the claims it would need to check in practice.

This project, the Bangla Live Fact-Checker (implemented as LiveFactChecker), addresses these two problems together: the static-knowledge weakness of LLM-only fact-checking, and the Bengali-language and multi-script gap in existing tools. The system is built around a single Python class, LiveFactChecker, which wraps a Groq client running the “openai/gpt-oss-120b”, “openai/gpt-oss-20b”, “qwen/qwen3.6-27b”-versatile model[8] together with a Tavily search client and a DuckDuckGo search tool as a fallback. The checking process occurs in order, with the first attempt being with the model “openai/gpt-oss-120b”. If this model is available, then it is used to complete the process. If this model is not available, then the second attempt involves “openai/gpt-oss-20b”, which is used if it is available. The third attempt involves checking “qwen/qwen3.6-27b” for availability. If all three attempts fail, then the process is unsuccessful. [**Note:** Due to the deprecation and

endpoint updates of earlier models (e.g., standard Llama-3.3-70b instances) on the Groq deployment platform, the updated API model endpoints (openai/gpt-oss-120b, qwen/qwen3.6-27b, etc.) were utilized to maintain system runtime and evaluation compatibility.]

A user is not required to type in any particular script or language: a claim can be entered in Bengali (Bangla script), in English, or in Banglish (for example, "padma te notun bridge hoyeche" instead of the Bangla-script equivalent). The submitted claim is first passed to the generate_multi_angle_queries method, which prompts the LLM to produce up to ten short Bengali and English search queries covering different phrasings and entities in the claim. The fetch_evidence method sends these queries to Tavily, deduplicates the results by URL, and falls back to DuckDuckGo if fewer than three pieces of evidence are found. Only after this evidence has been assembled does the verify_claim method ask the LLM to produce a verdict, using a prompt that explicitly instructs it to rely only on the retrieved evidence rather than on facts it may already know. The verdict is one of three fixed categories - সত্য/Real (Reported), মিথ্যা/Fake (Misleading), or Developing/Unverified - together with a short Bengali explanation. This is delivered to the user through a Streamlit interface (app.py), where a claim is typed into a text box, checked with a single button press, and the verdict, explanation, and a list of consulted sources are displayed in the browser.

This chapter situates the project within its broader research field, states the motivation behind building it, defines the problem it addresses, lists the objectives set for the system, and outlines the practical challenges encountered while building it. The chapters that follow describe, in turn, the development process (Chapter 2), the technology stack (Chapter 3), the system's architecture (Chapter 4), its implementation in detail (Chapter 5), its limitations and directions for future work (Chapter 6), and a concluding summary (Chapter 7).

1.2 Project Motivation

- **Primary Goal:** To build a fact-checking tool that gives ordinary Bengali-speaking readers a fast, evidence-based way to check a claim before sharing it further, rather than requiring them to wait for a manual fact-check to be published or to judge the claim's credibility unaided.

- **Interest in Applied AI:** To explore how large language models can be made more reliable by grounding their answers in retrieved, live evidence instead of static training data - an approach broadly known as retrieval-augmented generation (RAG) - and to apply that idea to a genuinely practical, real-world verification task rather than a purely academic benchmark.
- **Language Gap:** Most automated fact-checking tools and the research literature behind them are built for English. Bengali claims are underserved despite the scale of misinformation circulating in the language and despite Bengali being one of the most widely spoken languages in the world, which represents both a practical gap and a research opportunity.
- **Natural Input Habits:** Bangladeshi social media users frequently write in Banglish (Romanized, code-mixed Bengali) rather than Bangla script or formal English, particularly in casual contexts such as comments, group chats, and quickly shared posts - precisely the contexts in which misinformation tends to spread. A fact-checking tool that only accepts one script excludes a large share of real-world claims, so the system was designed from the outset to accept Bengali, English, or Banglish input interchangeably.
- **Real-Time Requirement:** News develops quickly, and a claim that was false yesterday may be confirmed today, or the reverse. A system relying only on an LLM's fixed training data cannot reflect this kind of change, so live search was treated as a core architectural requirement from the beginning of the project rather than as an optional add-on layered on afterward.
- **Transparency:** Rather than returning a verdict with no justification, the system was designed to show its sources alongside the verdict, so a user can inspect the evidence and judge its relevance themselves, instead of being asked to simply trust an unexplained label produced by a model.
- **Accessibility:** A lightweight Streamlit interface was chosen so the tool could be used directly through a browser without requiring the user to install anything, register an account, or have any technical background - lowering the barrier to actually using the tool in the moment a claim is encountered.

1.3 Problem Statement

- **Verification Speed:** Claims spread on social media far faster than they can be manually fact-checked by journalists or dedicated organisations, leaving an extended window during which misinformation circulates essentially unchecked and, in many cases, is never corrected for the majority of people who saw the original claim.
- **Language Barrier:** Existing automated fact-checking tools, datasets, and research are heavily concentrated in English, leaving Bengali-speaking users - a population of well over two hundred million people - with comparatively few automated options and forcing continued reliance on slower manual processes.
- **Static Knowledge Problem:** A language model used on its own for fact-checking is fundamentally limited to what it learned during training and has no mechanism for verifying claims about events that occurred after its training cut-off, which is precisely when a fact-checking tool is most needed.
- **Hallucination Risk:** Without grounding in retrieved evidence, an LLM can generate a plausible-sounding but factually incorrect verdict with the same confident tone it would use for a correct one, which is particularly dangerous in a fact-checking context where the entire purpose of the tool is to be trustworthy.
- **Lack of Source Transparency:** A verdict delivered without visible sources gives the user no way to judge its credibility, cross-check the reasoning, or investigate further if they are uncertain - reducing the tool to another unverifiable claim rather than a genuine aid to verification.
- **Query Formulation:** A single, literal search of a claim's exact wording frequently misses relevant news coverage, which may describe the same event using different phrasing, different transliteration of names, or an entirely different language than the one the claim was submitted in.
- **Evidence Reliability:** Search results returned by any web search API vary considerably in quality and relevance, and separating credible, on-topic reporting from unrelated, outdated, or low-quality results is not a trivial filtering problem, particularly when working across two languages and two scripts simultaneously.

1.4 Objectives

- **Live Evidence Retrieval:** Ground every verdict produced by the system in live web search results gathered at the moment a claim is checked, rather than in the language model's static, potentially outdated training data.
- **Multi-Angle Query Generation:** Use an LLM to automatically expand a single submitted claim into multiple bilingual (Bengali and English) search queries, so that retrieval coverage does not depend on the user having phrased the claim the same way a news source would.
- **Multi-Script Input Support:** Accept claims written in Bengali script, English, or Banglish (Romanized code-mixed Bengali) without requiring the user to specify which of the three forms they are using, so the tool matches how people actually type online.
- **Automated Verdict Classification:** Classify each claim as Real, Fake, or Developing based strictly on the retrieved evidence, using a fixed, constrained set of output categories rather than open-ended free text.
- **Bengali-Language Explanation:** Produce a concise, human-readable explanation of the verdict in Bengali, so the reasoning behind a verdict is accessible to the same audience the verdict itself is intended for.
- **Source Transparency:** Present the evidence and source links used to reach the verdict alongside the verdict itself, so a user can independently verify the basis for the system's conclusion.
- **Redundant Retrieval:** Use a primary search API (Tavily) together with an automatic fallback (DuckDuckGo), so that a single provider outage or an insufficient result set from one provider does not cause the whole verification request to fail.
- **Accessible Interface:** Deliver the system through a simple, browser-based Streamlit interface that requires no installation, account creation, or technical setup on the user's side.
- **Deduplication:** Avoid presenting the same source more than once to the user when multiple generated search queries happen to return overlapping results from the same article or page.

1.5 Challenges

- **API Latency:** A single fact-check involves one LLM call to generate search queries, up to ten live search calls against Tavily and potentially DuckDuckGo, and a second LLM call to produce the final verdict, all of which must complete before the user sees a result - keeping this sequence responsive enough for interactive, one-claim-at-a-time use required careful attention to how many calls were made and in what order.
- **Prompt Reliability:** Constraining the LLM to output a verdict in a fixed, machine-parseable format, and to reason only from the supplied evidence rather than from its own training data, is not something a language model does reliably on the first attempt; it required several rounds of iterative prompt design, discussed in detail in Chapter 2.
- **Evidence Quality Control:** Search results returned by Tavily or DuckDuckGo are not uniformly reliable, and the system has no independent, separate mechanism to score source credibility beyond whatever judgement the LLM itself applies when reading the retrieved content.
- **Bilingual Query Generation:** Generating search queries that work well in both Bengali and English simultaneously, including correct transliteration of names, places, and institutional terms between the two, needed specific and repeated prompt tuning rather than a single generic instruction.
- **Interpreting Banglish Input:** Romanized, code-mixed Bengali does not follow a fixed or standardised spelling convention - the same word can be typed several different ways by different users - so reliably interpreting the intended meaning of a Banglish claim before generating search queries added a layer of ambiguity that a purely Bangla-script or purely English system would not have to handle.
- **Graceful Degradation:** Ensuring the system still returns a usable, if less confident, answer when the primary search provider fails outright or returns too few results, rather than simply erroring out or leaving the user with no response at all.
- **Error Handling and Debuggability:** Balancing a smooth, uninterrupted user experience - not surfacing raw Python exceptions or stack traces in the Streamlit interface - against the practical need to still be able to log and diagnose failures

during development, a tension that shaped several of the design decisions described in Chapter 2.

- **Scope of Verification:** Deciding how the system should respond when there is genuinely no live news coverage of a claim at all, rather than forcing the model into a Real or Fake verdict in a situation where the available evidence simply does not support either conclusion.

CHAPTER 2

SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC)

2.1 Introduction

The Software Development Life Cycle (SDLC) is the structured sequence of phases followed to design, build, and maintain a software product. It exists as a formal concept in software engineering because building software without any structure tends to produce systems that are hard to test, hard to extend, and difficult for anyone other than the original author to reason about.[13] Different SDLC models organise these phases in different ways, and the choice of model is not a cosmetic one: it shapes how requirements are gathered, how much rework is tolerated, and how quickly a team can respond when something learned midway through development contradicts an earlier assumption. Fig. 2.1 shows the general phases - planning, requirement analysis, design, implementation, testing, and deployment - that most SDLC models are built around, whatever order or repetition they impose on those phases.

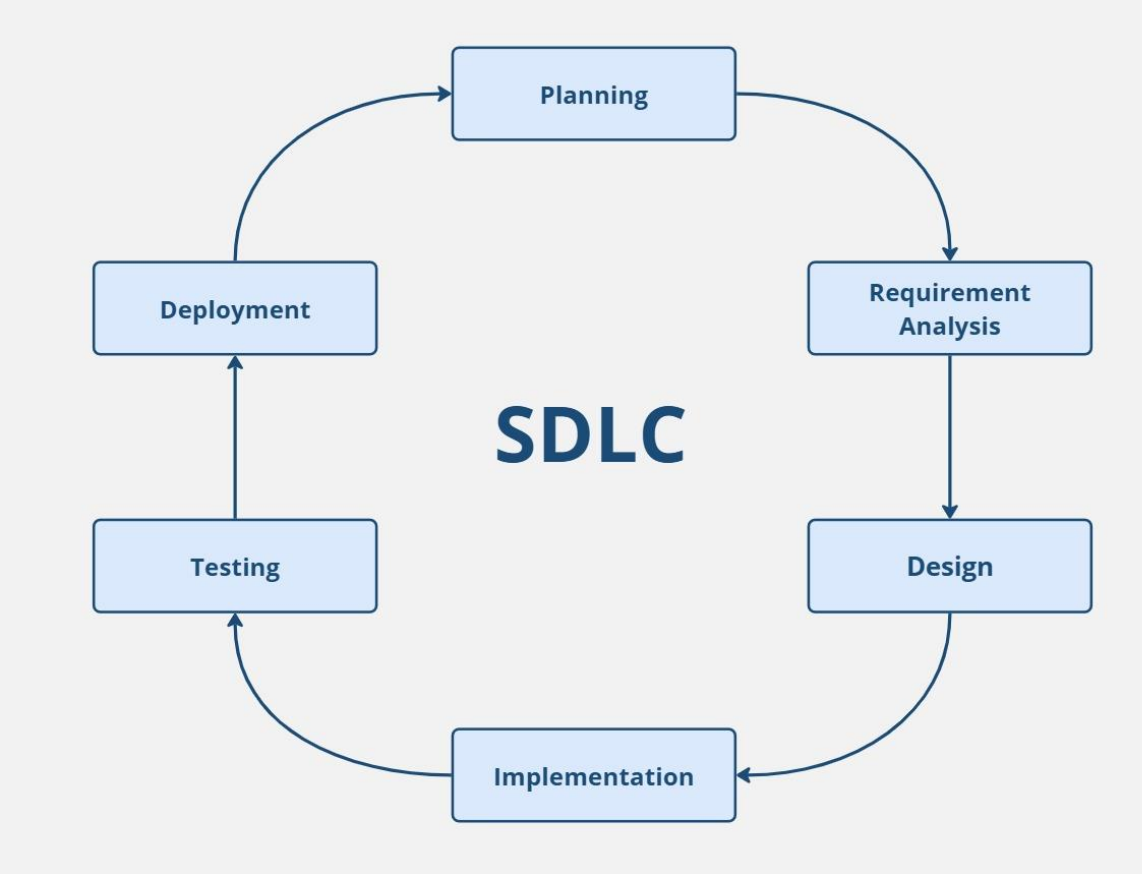


Fig. 2.1: SDLC

The remainder of this chapter first reviews the SDLC models available and explains why a strictly sequential model does not describe how LiveFactChecker was actually built, then walks through which model does describe it, and why that distinction matters for a system with a significant AI component rather than a purely deterministic, rule-based one.

2.2 Software Development Life Cycle Models

Several SDLC models are used in practice, each suited to a different kind of project, and each encoding a different assumption about how well the requirements are understood before implementation starts:

- Waterfall Model
- V-Model
- Iterative Model
- Spiral Model

The Waterfall Model executes each phase exactly once, in strict sequence, and assumes requirements are fully known before implementation begins; it is well suited to projects where the specification is fixed and stable, such as systems built against a rigid regulatory or contractual requirement. This does not describe how the fact_checker.py backend was actually written. Decisions such as how many search queries generate_multi_angle_queries should request from the LLM (the final version caps this at ten, via unique_queries[:10]), what should happen when the Tavily client fails to initialise (handled by wrapping the TavilyClient(api_key=...) call in a try/except so self.tavily can safely be None), and exactly what instructions the verify_claim prompt needed to stop the LLM from answering out of its own memory instead of the retrieved evidence - none of these were known in advance, and none of them could realistically have been specified correctly on paper before the system existed to test them against. Each was arrived at by running the system, observing incorrect or incomplete behaviour, and revising the corresponding function. For this reason, the project followed the Iterative Model rather than the Waterfall Model.

2.3 Iterative Model Design

In the Iterative Model, a working version is produced early and then refined over repeated cycles, with each cycle adding functionality or fixing problems observed in the previous one, as shown in Fig. 2.2. Rather than attempting to specify the whole system's behaviour

up front, each cycle narrows the gap between the current implementation and the desired behaviour based on evidence from running the system, not on assumptions made in advance. LiveFactChecker went through several such cycles, summarised below in the order they occurred:

- **Iteration 1 - Single-query retrieval:** The earliest working version passed the claim text directly into a single search call inside what became fetch_evidence, with no query expansion. Testing showed this missed relevant news coverage whenever the claim was phrased differently from how it was actually reported - for example, a claim naming a person informally while news coverage used their full title, or a claim using Banglish spelling that did not match how the topic was indexed by the search provider.
- **Iteration 2 - Multi-angle query generation:** The generate_multi_angle_queries method was added, prompting the LLM to produce up to ten Bengali and English queries from the same claim, so that a claim's key entities and phrasing could be searched from several different angles rather than just one. A deterministic backup path was also added - splitting the claim into its first three and last three words (words[:3] and words[-3:]) - so the method still returns usable queries even if the LLM call itself fails, ensuring the pipeline degrades gracefully rather than stopping outright.
- **Iteration 3 - Redundant search with fallback:** Relying on Tavily alone occasionally produced too little evidence, or failed outright when the API was unavailable or the key was misconfigured. The fetch_evidence method was revised to fall back to self.ddg.run(q) (DuckDuckGo) whenever fewer than three evidence items had been collected, and a seen_urls set was introduced to drop duplicate sources returned by overlapping queries, since several of the generated queries frequently pointed back to the same underlying article.[17]
- **Iteration 4 - Constrained verdict prompting:** Early versions of the verify_claim prompt sometimes let the LLM answer from what it already knew about a topic rather than the evidence just retrieved, effectively defeating the purpose of the retrieval step. The Bengali prompt was rewritten with explicit numbered rules forbidding this, and constrained to output exactly two labelled lines, VERDICT:

and EXPLANATION:, so app.py could reliably parse the response without needing more elaborate natural-language parsing logic on the front end.

- **Iteration 5 - Interface refinement:** The Streamlit front end (app.py) was iterated on separately from the backend logic - adding the custom CSS theme, mapping the parsed verdict to st.success / st.error / st.warning based on whether the string সত্য or মিথ্যা appears in it, and wrapping the evidence summary in a collapsible st.expander so the verdict stays the visual focus of the page while the supporting sources remain available without cluttering the initial view.

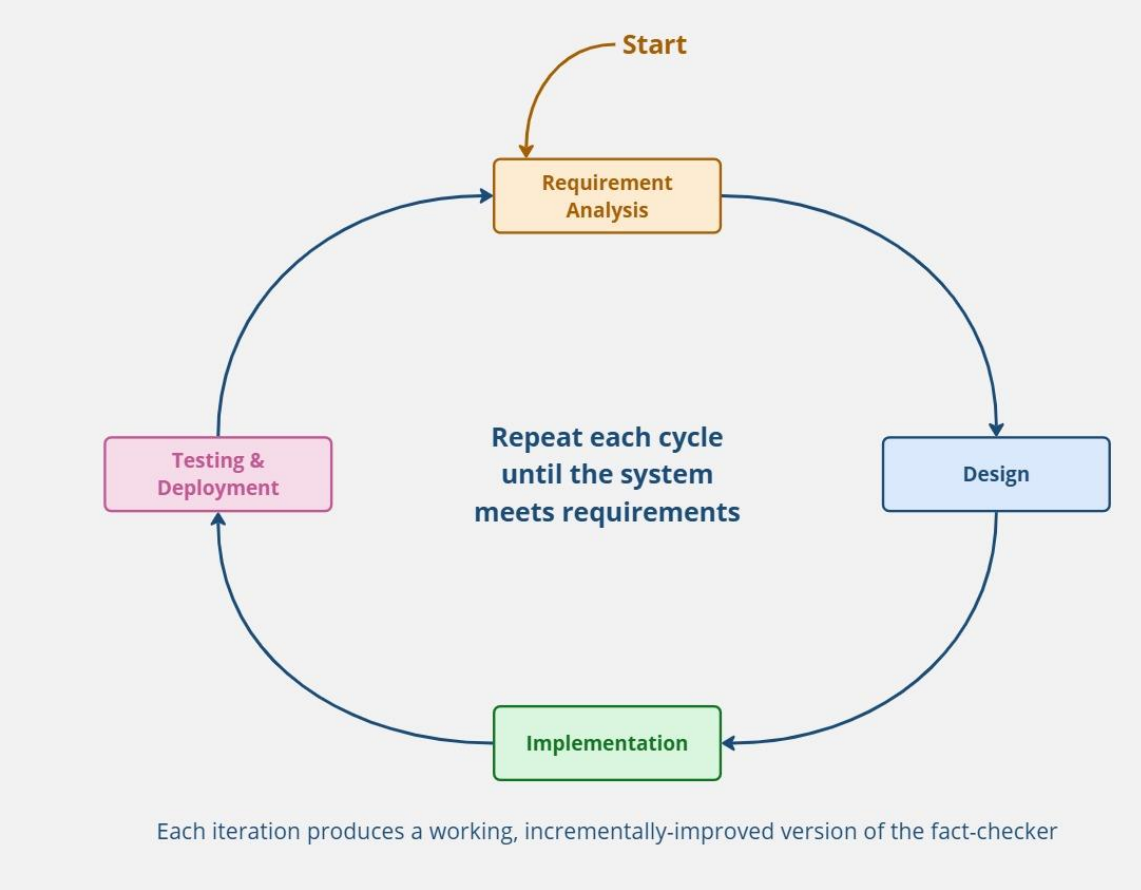


Fig. 2.2: Iterative Model

2.4 Why the Iterative Model is Better for LiveFactChecker

- **Evolving Requirements:** Concrete parameters in the code - the query cap in unique_queries[:10], the max_results=2 passed to Tavily's search() call, and the minimum-evidence threshold of 3 that triggers the DuckDuckGo fallback - were all tuned after observing real runs, not decided before writing the first line of

fact_checker.py. A Waterfall approach would have required committing to these numbers in a design document before any evidence existed to justify them.

- **Early Feedback:** Because a working verify_claim existed early, later iterations could be tested against real claims immediately, rather than discovering prompt weaknesses only once the whole system was 'finished' and every component had to be debugged simultaneously.
- **Incremental Risk Reduction:** The broad except Exception: pass blocks scattered through generate_multi_angle_queries and fetch_evidence exist because specific failures - a Groq timeout, an empty Tavily response, a malformed API response - were observed one at a time during earlier iterations and handled as they appeared, rather than every possible failure mode being anticipated correctly in advance, which is rarely realistic for a system built on top of third-party APIs.
- **Compatibility with Prompt-Based Development:** The VERDICT / EXPLANATION output format in verify_claim is the result of several rewrites of the same prompt, each correcting a specific failure mode observed in the previous version; this trial-and-error style of refinement is what the Iterative Model is designed for and what a single-pass Waterfall approach, which assumes the specification is correct the first time, does not accommodate well.
- **Flexibility in Tooling:** Swapping in a second search provider (DuckDuckGoSearchRun) and adding a resilience layer around TavilyClient were both changes made to an already-working system in response to observed reliability gaps, not part of an original design document - a pattern of adaptation the Iterative Model treats as normal rather than as a deviation from plan.

CHAPTER 3

DEVELOPMENT TOOLS AND TECHNOLOGIES

3.1 Introduction

Unlike a conventional CMS or CRUD-based application, LiveFactChecker does not maintain a persistent database of its own. Its technology stack is built instead around orchestrating calls to a large language model and two live search providers, then presenting the result through a lightweight interface. This is a deliberately narrow stack: the system does not need user accounts, stored history, or a schema of any kind, so the usual back-end concerns of a data-driven application - migrations, query optimisation, referential integrity - simply do not arise. What replaces them is a different kind of complexity: coordinating several external API calls reliably, in the right order, within a time budget the user will tolerate. This chapter documents the languages, libraries, and APIs used to build the system, and explains why each was chosen over available alternatives, before Chapter 4 turns to how these pieces are wired together architecturally and Chapter 5 walks through the resulting code in detail.

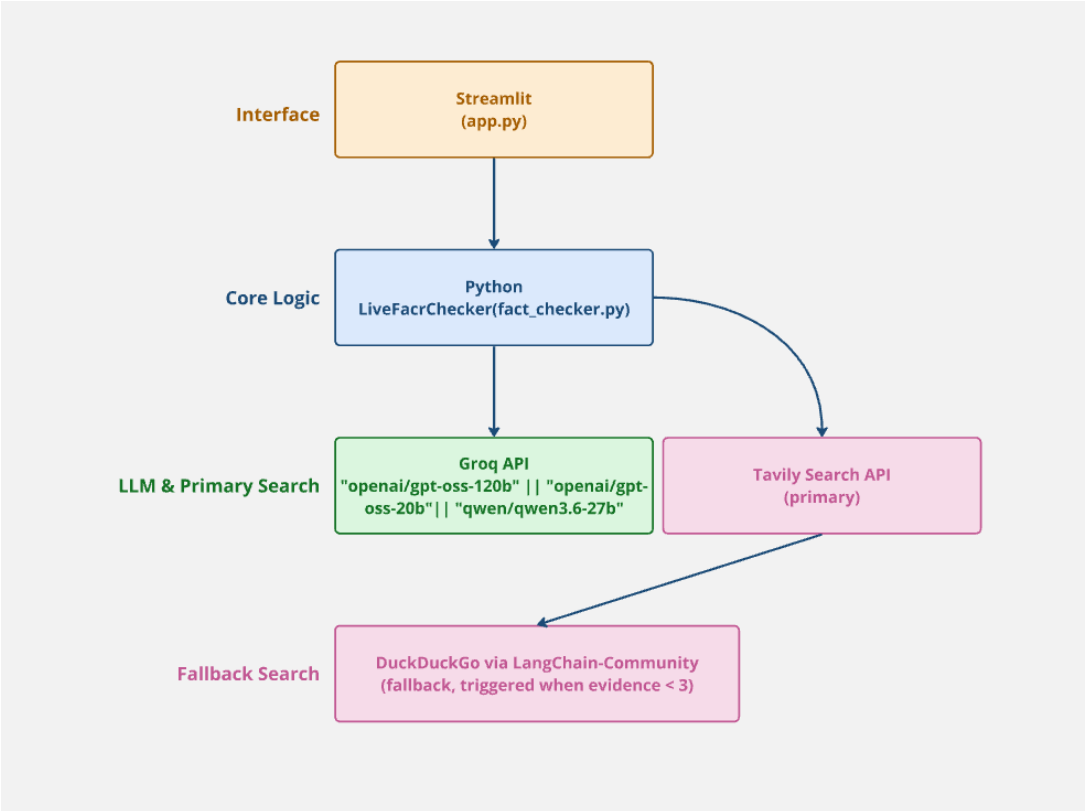


Fig. 3.1: LiveFactChecker Technology Stack

3.2 Programming Language and Core Libraries

- **Python:** Both the backend (LiveFactChecker class in fact_checker.py) and the front end (app.py) are written entirely in Python. This was less a deliberate architectural choice than a practical one - every library the project depends on (groq, tavily-python, streamlit, langchain-community) ships as a first-class Python package, so writing the whole system in one language avoided any inter-process or API-boundary overhead between the UI and the fact-checking logic. It also meant a single virtual environment and a single requirements.txt were sufficient to reproduce the whole system, rather than needing to coordinate a separate runtime for the front end and the back end.

- **Groq (LLM Inference API):** The groq.Groq client connects to Groq's hosted inference endpoint running “openai/gpt-oss-120b”, “openai/gpt-oss-20b”, “qwen/qwen3.6-27b”-versatile[8], [11]. The model is called twice per fact-check request: once inside generate_multi_angle_queries() to expand a single claim into up to ten search queries, and once inside verify_claim() to produce the final VERDICT/EXPLANATION pair from the retrieved evidence. Groq was chosen specifically for inference speed - since a single fact-check already involves up to ten sequential search calls on top of two LLM calls, a slower-hosted model would have made the tool feel unusably slow for an interactive, claim-at-a-time tool, where a user is actively waiting in a browser tab for the verdict rather than submitting a batch job.

  Note: Due to the deprecation and endpoint updates of earlier models (e.g., standard Llama-3.3-70b instances) on the Groq deployment platform, the updated API model endpoints (openai/gpt-oss-120b, qwen/qwen3.6-27b, etc.) were utilized to maintain system runtime and evaluation compatibility

- **Tavily (Search API):** TavilyClient is used as the primary evidence source inside fetch_evidence(), called per query with search_depth="advanced" and max_results=2[12]. Tavily was chosen over a raw scraping approach or a general search API because it is purpose-built for LLM pipelines - it returns pre-parsed title, content, and url fields directly, removing the need for the project to handle HTML parsing, boilerplate removal, or content extraction itself, all of which would

otherwise have been a substantial extra engineering surface for a project of this scope.

- **DuckDuckGo Search (via langchain-community):** DuckDuckGoSearchRun serves as the fallback path, triggered whenever Tavily returns fewer than three evidence items, or when the TavilyClient fails to initialize at all (handled by the try/except around the client construction in __init__). Its main advantage is that it requires no API key, making it a free, always-available backup that keeps the system functional even if the Tavily subscription lapses or the key is misconfigured - the trade-off is that it returns a single block of summarized text rather than structured results with clean URLs, which is why the front end renders DuckDuckGo-sourced evidence items differently (without a clickable source link). [7]

3.3 Front-End Framework

- **Streamlit:** app.py is built entirely in Streamlit, chosen because it allows a functioning, interactive browser UI to be written in pure Python, without a separate HTML/CSS/JS build pipeline, a bundler, or a JavaScript front-end framework to learn and maintain alongside the Python backend. Custom styling - the animated gradient background, glassmorphism header card, and themed alert boxes - is injected through a single st.markdown(..., unsafe_allow_html=True) block, layered on top of Streamlit's native components (st.text_area, st.button, st.expander, st.spinner) rather than replacing them. This kept state handling (button clicks, spinner visibility, expander toggling) inside Streamlit's own rerun model instead of requiring custom JavaScript or a client-side state management layer.
- **st.cache_resource:** The LiveFactChecker instance is created once, in load_checker(), decorated with st.cache_resource. This was necessary because Streamlit reruns the entire script on every widget interaction - without caching, a fresh Groq and TavilyClient object would be instantiated on every keystroke or click, which is both wasteful and, in Tavily's case, would repeat the client's initialization check unnecessarily, adding latency to every single interaction rather than only to the actual fact-check request.

3.4 Dependency Management

The project's requirements.txt lists five direct dependencies: streamlit, groq, tavily-python, langchain-community, and duckduckgo-search. The list is deliberately short - there is no ORM, no web framework beyond Streamlit itself, and no database driver, which reflects the stateless, single-session design discussed in Chapter 2: each fact-check is a self-contained request-response cycle with nothing persisted between runs. Keeping the dependency surface small also reduces the amount of third-party code that has to be trusted and kept up to date, which matters for a project maintained by a small student team rather than a dedicated engineering organisation.

3.5 Development Environment and Configuration

The system was developed and run locally, with GROQ_API_KEY and TAVILY_API_KEY currently defined as hardcoded string literals at the top of fact_checker.py rather than loaded from environment variables. This is a known limitation of the current build rather than an intentional design decision, and moving credential handling to environment variables (e.g., via os.environ or a .env file) is listed as a near-term improvement in Chapter 6. During development, this configuration was sufficient because the code was run only on local machines under the developers' control, but it would need to change before the project could be shared publicly or deployed to a server without exposing the credentials alongside the source code.

3.6 System and Browser Requirements

Running LiveFactChecker requires Python 3.9 or later, an active internet connection - since every fact-check depends on live calls to the Groq and Tavily APIs - and a modern web browser capable of rendering Streamlit's client (Chrome, Firefox, or Edge). No GPU or specialized hardware is required, since inference is delegated entirely to Groq's hosted endpoint rather than run locally, which also means the system can run comfortably on an ordinary laptop without the memory or compute demands that running a 70-billion-parameter model locally would otherwise impose.

CHAPTER 4

SYSTEM ARCHITECTURE AND DESIGN

4.1 Introduction

LiveFactChecker has no persistent database, so its design cannot be documented the way a CRUD application's is - there are no tables, primary keys, or foreign keys to diagram, and no entity-relationship model to draw in the conventional sense. What the system does have is a well-defined pipeline: a claim enters as plain text, passes through a fixed sequence of transformations, and a verdict comes out the other end. Architecturally, this makes LiveFactChecker closer to a request-processing pipeline or an orchestration layer over external services than to a traditional multi-tier application with a data layer of its own. This chapter documents that architecture from three complementary angles: the data objects that flow through the pipeline and the functions that produce them (Section 4.2), the use cases available to the user and the external services the system depends on (Section 4.3), and a data-flow diagram showing exactly what information crosses the boundary between LiveFactChecker and each external API (Section 4.4).

4.2 Data Objects and Flow

Rather than relying on persistent entities that are permanently stored in a database, LiveFactChecker uses four temporary in-memory data objects to manage the fact-checking process. Each of these objects is generated by a specific method within the LiveFactChecker class and is then passed to the next stage of the processing pipeline. In this way, the system can move information from one stage to another without requiring the intermediate results to be saved permanently.[16] The objects exist only for the duration of a single fact-checking request and are not maintained after the final verdict has been produced. Once the verification process is completed and the result is returned to the Streamlit interface, all intermediate objects are discarded from memory. When a new fact-checking request is received, the system creates a fresh set of objects and begins the processing sequence again. This approach keeps each request independent, reduces unnecessary data persistence, and provides a clean processing state for every verification task. The complete flow of these objects through the system is illustrated in Fig. 4.1.

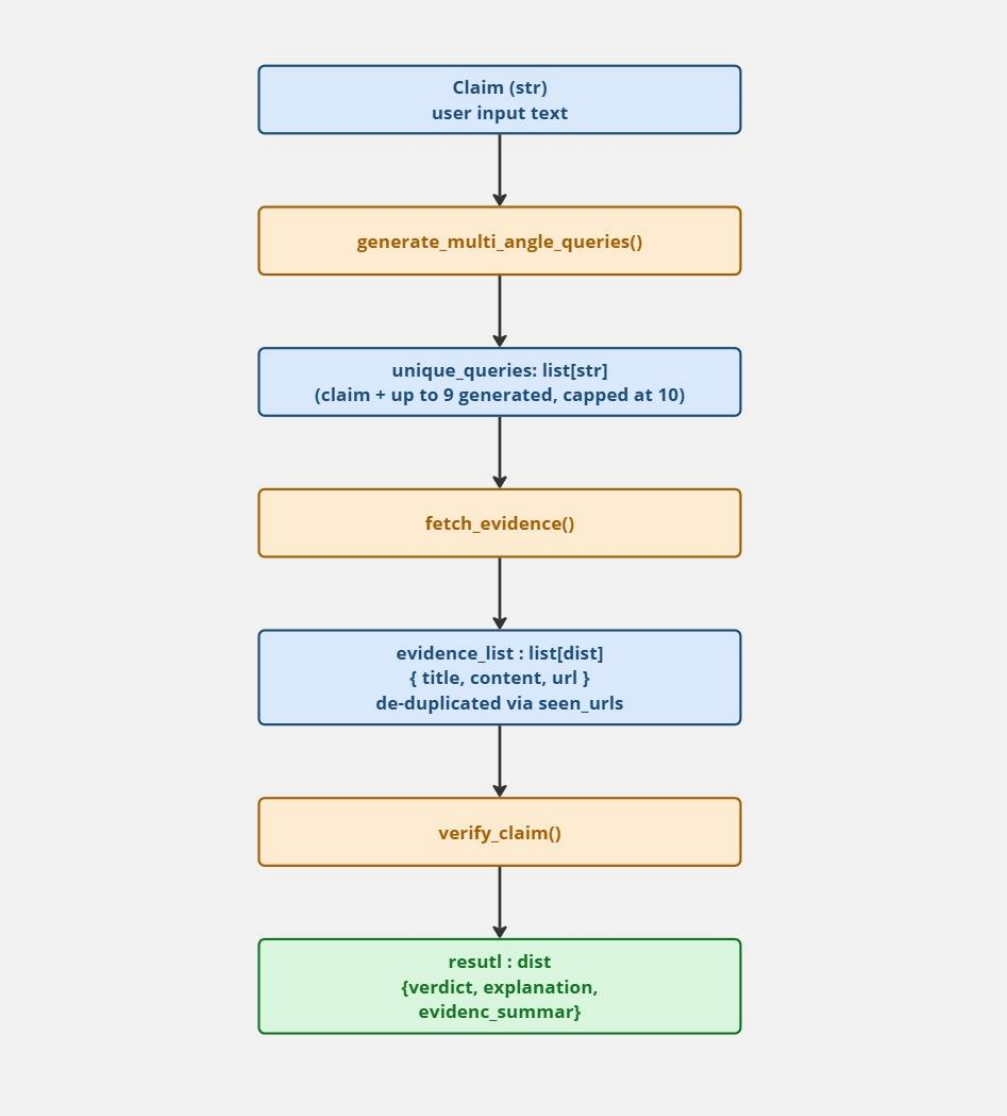


Fig. 4.1: Data Objects and Flow through LiveFactChecker

- **Claim:** A plain Python string, taken directly from the Streamlit text area in app.py after a .strip() call. It may be written in Bengali, English, or Banglish, and no language tag or metadata is attached to it - the burden of interpreting which script or language is in use falls entirely on the LLM at the next stage rather than on any explicit pre-processing step.
- **unique_queries:** A list of strings produced by generate_multi_angle_queries(). It always begins with the original claim itself, followed by up to nine queries generated by the LLM, plus two backup queries built by splitting the claim's own words if the LLM call fails. Duplicates are removed and the list is capped at ten entries with unique_queries[:10], which bounds the worst-case number of downstream search calls and keeps request latency predictable.

- **evidence_list:** A list of dictionaries built inside fetch_evidence(), one per unique search result, each holding title, content, and url keys taken from the Tavily response (or a synthesised title for DuckDuckGo results, which do not include a URL). A seen_urls set prevents the same source appearing twice when different queries return overlapping results, which happens frequently since several of the generated queries are variations on the same underlying event.
- **result:** The dictionary returned by verify_claim(), holding verdict, explanation, and evidence_summary. This is the only object that reaches the Streamlit interface, where its three fields are mapped directly onto the verdict banner, the explanation text, and the contents of the evidence expander - the interface never sees the raw evidence_list or unique_queries objects directly, only this final, already-reasoned-over result.

4.3 Use Case Diagram

Fig. 4.2 shows the use cases available to the User actor, alongside the three external service actors the system depends on to complete a fact-check. Unlike the CMS-style systems this chapter format is usually applied to, LiveFactChecker has only one human actor - there is no separate administrator role, since the system has no content to moderate, no user accounts to manage, and no persistent state that would require an administrative interface to control.

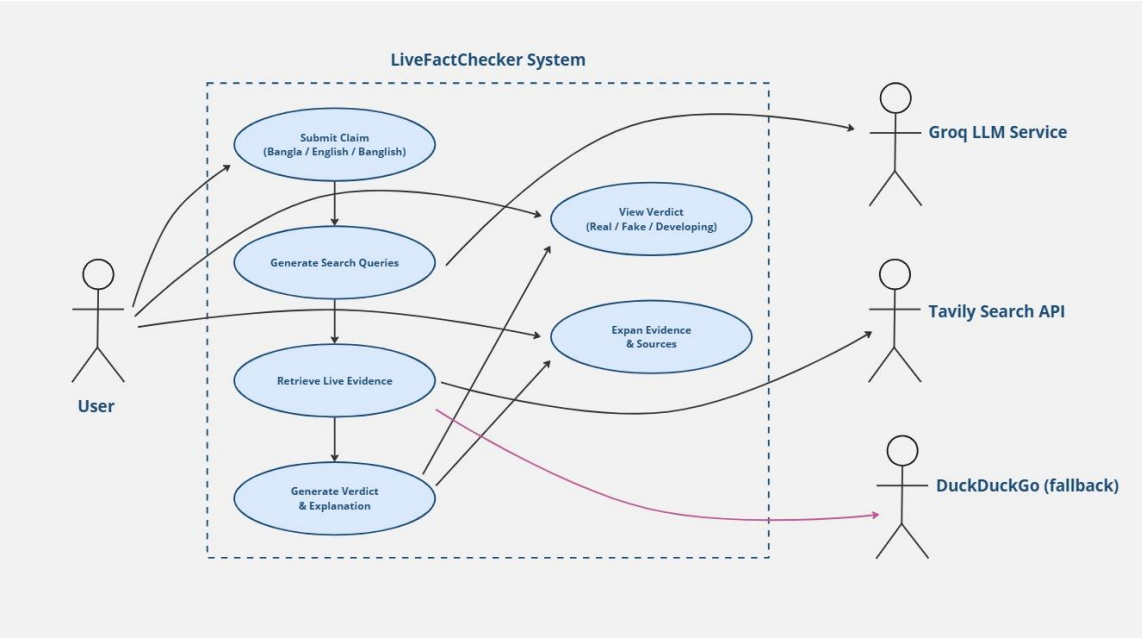


Fig. 4.2: Use Case Diagram

- **User:** Submits a claim, views the resulting verdict, and can optionally expand the evidence panel to inspect the retrieved sources. These map directly to the claim_input text area, the verdict success/error/warning banner, and the st.expander block in app.py - every use case available to the user corresponds to a specific, identifiable widget in the interface, with no hidden functionality.
- **Groq LLM Service:** Participates in two use cases - generating search queries and generating the verdict - corresponding to the two separate calls to self.client.chat.completions.create() in fact_checker.py, each configured with its own prompt and temperature setting as described in Chapter 5.
- **Tavily Search API:** Participates in the evidence-retrieval use case as the primary source, queried once per generated search query, making it the service the system depends on most heavily in terms of call volume.
- **DuckDuckGo (fallback):** Participates in evidence retrieval only when Tavily's results are insufficient, reflecting the conditional (if len(evidence_list) < 3) fallback path in fetch_evidence() - under normal operation with a healthy Tavily connection, this actor may not be exercised at all for a given request.

4.4 Data-Flow Diagram

Fig. 4.3 is a context-level data-flow diagram treating the whole of LiveFactChecker as a single process (0.0) and showing the data that crosses its boundary with the user and with each external API. This single-process view is appropriate here because, internally, the system has no branching workflow visible to the user - every claim follows the same sequence of internal calls, so a Level-1 decomposition would only restate the method breakdown already given in Fig. 4.1 without adding new information about how data actually moves through the system.

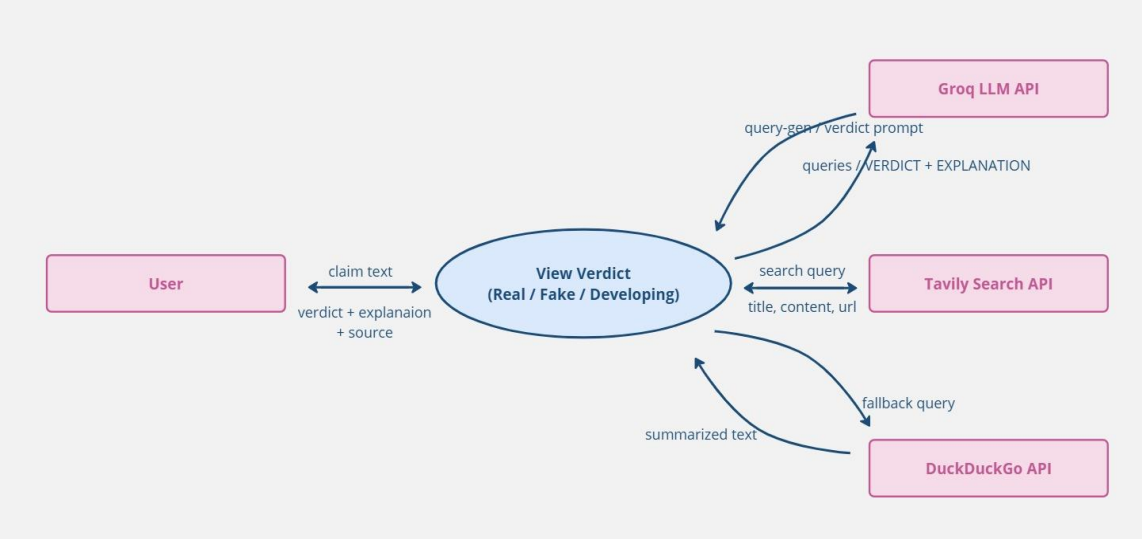


Fig. 4.3: Data-Flow Diagram (Level 0 - Context Diagram)

Two details in this diagram are worth calling out explicitly. First, the arrow between LiveFactChecker and Groq LLM API is used twice for two structurally different exchanges - a query-generation prompt paired with a plain list of queries, and a verdict prompt paired with the constrained VERDICT/EXPLANATION response - but both are shown on the same pair of arrows since they share the same external boundary and the same underlying API endpoint, even though the prompt content and expected response shape differ substantially between the two calls. Second, the DuckDuckGo arrow is conditional rather than unconditional: it is only exercised on a subset of requests, unlike the always-executed Tavily exchange immediately above it, which means the diagram represents the union of all possible data flows rather than the flow for any single fixed request.

CHAPTER 5

LIVEFACTCHECKER: IMPLEMENTATION

5.1 Introduction

This chapter walks through how the architecture described in Chapter 4 is actually implemented across the two files that make up the system: fact_checker.py, which holds the LiveFactChecker class and all retrieval and reasoning logic, and app.py, which implements the Streamlit interface. The backend is covered method by method in the order a claim actually passes through them - initialisation, query generation, evidence retrieval, and verdict generation - so that the description follows the same sequence a real request follows at runtime rather than an arbitrary file order. The front end is then covered component by component, from page configuration through to result display, and the chapter closes with an end-to-end walkthrough of the running application, illustrated with screenshots of real verdicts the system produced.

5.2 Back-End Implementation (fact_checker.py)

5.2.1 Initialization

The LiveFactChecker.__init__ method sets up three clients: a Groq client authenticated with GROQ_API_KEY, a DuckDuckGoSearchRun tool from langchain_community, and a TavilyClient authenticated with TAVILY_API_KEY. The Tavily client is constructed inside a try/except block, so that if construction fails - for example, because of an invalid key, a network issue at start-up, or a change in the Tavily SDK's expected arguments - self.tavily is set to None instead of raising an exception and crashing the whole application before it has even served its first request. Every later call site that uses self.tavily checks for this with an if self.tavily: guard before calling it, which means the rest of the class never needs to worry about whether the Tavily client actually exists; it simply behaves as though Tavily is temporarily unavailable and lets the fallback logic in fetch_evidence take over.

5.2.2 Multi-Angle Query Generation

generate_multi_angle_queries(claim) is the first step in handling any claim. It starts the query list with the claim itself, then sends a Bengali-language prompt to the LLM asking for eight to ten short, high-precision search queries in both Bengali and English, explicitly

instructing the model to break the claim down into its key entities, names, and current designations, and not to use site: filters. The call uses temperature=0.2 - a small amount of randomness so repeated angles are not simply copies of each other, while still staying low enough that the queries remain grounded in the claim's actual content rather than drifting into unrelated territory.[18] The response is split by newline and each non-empty line is appended to the query list, giving the method a simple, predictable way to turn the model's free-text output into a list the rest of the pipeline can consume.

If this LLM call raises an exception for any reason - a network timeout, a malformed response, a rate-limit error from Groq - the method does not fail outright: the except Exception: pass block simply skips straight to a deterministic backup step, where the claim's own words are split and the first three and last three words are added as two additional fallback queries. This guarantees the method always returns at least a small set of usable queries, even in the worst case where the LLM is completely unreachable. Finally, the list is deduplicated (skipping anything shorter than two characters) and truncated to unique_queries[:10] before being returned, guaranteeing the caller never receives more than ten queries regardless of how many the LLM produced, which in turn bounds how many search calls fetch_evidence will need to make.

5.2.3 Evidence Retrieval

fetch_evidence(claim) takes the query list from generate_multi_angle_queries and, for each query, calls self.tavily.search() with search_depth="advanced" and max_results=2 inside a try/except so that a single failed query does not abort the whole loop - if one query fails, the method simply moves on to the next one rather than losing all the evidence gathered so far. Each result's url, title, and content are read from the response; if the url has not already been seen (tracked via a seen_urls set), the result is appended to evidence_list, which is the mechanism that prevents the same article being counted, and shown to the user, multiple times just because several different generated queries happened to surface it.

After all queries have been tried against Tavily, the method checks len(evidence_list) < 3. If fewer than three pieces of evidence were collected - either because Tavily returned little for this particular claim, or because self.tavily was None to begin with - it falls back to DuckDuckGo, running the first three queries through self.ddg.run(q) and keeping any result longer than thirty characters, labelled with a generated title rather than a real one since DuckDuckGoSearchRun returns a single block of summarized text rather than structured,

per-source results with URLs. This threshold of three was chosen empirically during the iterative development process described in Chapter 2, as a balance between giving Tavily a fair chance to return sufficient evidence and not leaving the user with too thin a basis for a verdict.

Finally, the evidence list is formatted into a single Markdown string - a numbered list where each entry shows the source title as a clickable link (when a URL is available) followed by its content on the next line - and this formatted string, not the raw list, is what gets passed into the verdict prompt in the next step. Formatting the evidence this way serves two purposes at once: it gives the LLM a clean, readable block of text to reason over, and it is the exact same string later shown to the user inside the evidence expander, so there is no risk of the explanation being based on a different view of the evidence than what the user is shown.

5.2.4 Verdict Generation

verify_claim(claim) first calls fetch_evidence to obtain the formatted evidence string, then constructs a Bengali-language prompt that embeds both the claim and the evidence directly. The prompt lays out four numbered rules for the model: do not use your own internal training memory or past knowledge; base every decision only on the live search data provided; if the claim concerns a person's recent statement or current position, cross-check it specifically against the live evidence; and if there is no information available about a person or position, do not substitute old or outdated information. These four rules exist specifically to counteract the static-knowledge and hallucination problems described in Chapter 1 - without them, the model has a strong tendency to answer from what it already knows rather than from what was actually retrieved. The model is restricted to exactly three verdict categories - সত্য/Real (Reported), মিথ্যা/Fake (Misleading), and ⚡ Developing/Unverified - and instructed to output only two labelled lines, VERDICT: and EXPLANATION:, with the explanation limited to two or three lines of Bengali so the result stays readable at a glance rather than becoming a long, unstructured essay.

This call uses temperature=0.0, unlike the query-generation call, because a fact-checking verdict should be as deterministic and reproducible as possible rather than varying between runs on the same input - a fact-checking tool whose answer changes each time the same claim is submitted would undermine the trust the tool is trying to build. The response is parsed line by line, looking for lines starting with VERDICT: and EXPLANATION:, with

fallback default strings used if either label is missing from the model's output, so a malformed response degrades gracefully into a generic message rather than crashing the request. The method returns a dictionary with verdict, explanation, and evidence_summary keys; if the LLM call itself raises an exception, a separate except block returns an error-flagged dictionary instead of letting the exception propagate up into the Streamlit interface, where an unhandled Python exception would otherwise be shown directly to the end user.

5.3 Front-End Implementation (app.py)

5.3.1 Page Configuration and Styling

app.py begins with st.set_page_config, setting the page title, a search icon, and a centered layout, which together determine how the tab and initial page frame appear before any content has loaded. A large block of custom CSS is then injected through st.markdown(..., unsafe_allow_html=True), overriding Streamlit's default styling: an animated diagonal gradient background on .stApp, a glassmorphism-style header card with a blurred background and soft border, styled text-area and button elements, and recoloured alert boxes and expander panels so that result text remains clearly readable against the dark theme. This CSS block is purely presentational - it does not add any new interactive behaviour beyond what Streamlit's own components already provide, and could in principle be removed entirely without changing what the application does, only how it looks.

5.3.2 Session Setup and Input

A cached LiveFactChecker instance is created once via a load_checker() function decorated with @st.cache_resource, so the Groq, Tavily, and DuckDuckGo clients are not re-constructed on every Streamlit rerun - a detail that matters because Streamlit's execution model re-runs the entire script from top to bottom on almost every user interaction, and without this caching decorator the application would silently re-authenticate all three API clients on every keystroke. The claim itself is collected through a single st.text_area with an empty placeholder, followed by a full-width button labelled ফ্যাক্ট চেক করুন ("Check the fact"), keeping the entire input surface of the application to exactly one field and one action.

5.3.3 Handling the Check and Displaying Results

When the button is pressed, the app first checks whether the input is empty and shows a warning if so, preventing a wasted API call for a request that could not produce a meaningful verdict anyway. Otherwise, it enters a st.spinner block with the message ব্যাকগ্রাউন্ডে ১০টি ভিন্ন কোণ থেকে গভীর অনুসন্ধান চলছে… ("Searching from ten different angles in the background...") and calls checker.verify_claim(claim_input) synchronously - the user-facing spinner is the only feedback given during the several seconds this call can take, since the underlying pipeline described in Section 5.2 runs as a single blocking sequence of LLM and search calls rather than as a set of parallel or streamed operations.

Once a result dictionary comes back, its verdict string is checked with two plain substring tests: if সত্য ("true") appears in it, the verdict is shown in a green st.success box; if মিথ্যা ("false") appears, it is shown in a red st.error box; anything else - including the Developing/Unverified case - falls through to a yellow st.warning box. The explanation is shown separately in an st.info box, and the raw evidence_summary Markdown string is rendered inside a collapsible st.expander labelled সংগৃহীত লাইভ নিউজ পোর্টাল ও রেফারেন্স লিংক দেখুন ("View collected live news portals and reference links"), so the verdict itself stays the visual focus of the page while the supporting sources remain one click away rather than being forced on the user immediately.

5.4 End-to-End Walkthrough

This section presents screenshots of the deployed Streamlit interface while verifying real claims, demonstrating how the application operates end to end as described in Sections 5.2 and 5.3.

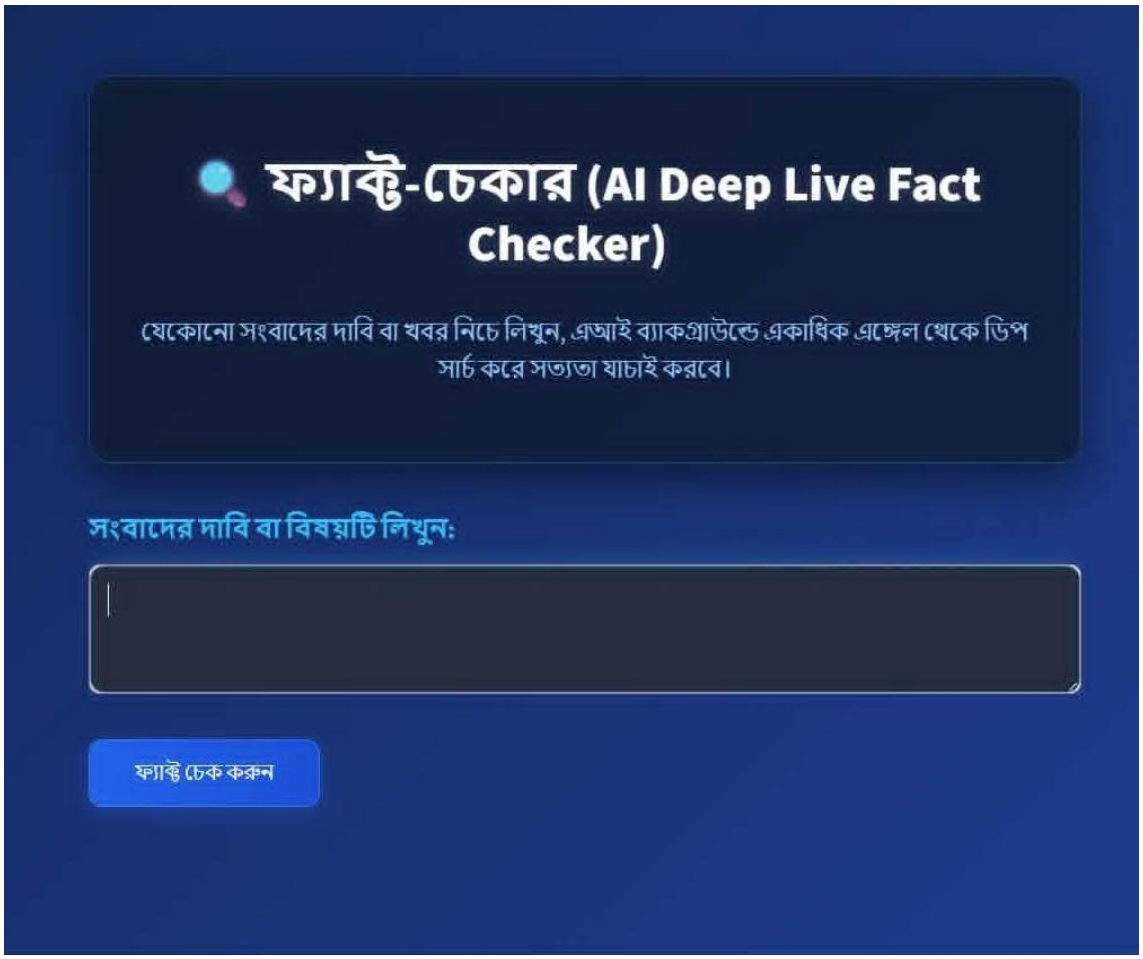


Fig. 5.1: Application Home Page

The home page, shown in Fig. 5.1, is the state of app.py immediately after streamlit run app.py is launched - before load_checker() has been exercised by any user action. The header card, the single text input, and the check button are the entire interface surface at this point, reflecting the deliberately minimal design discussed in Section 5.3.1.

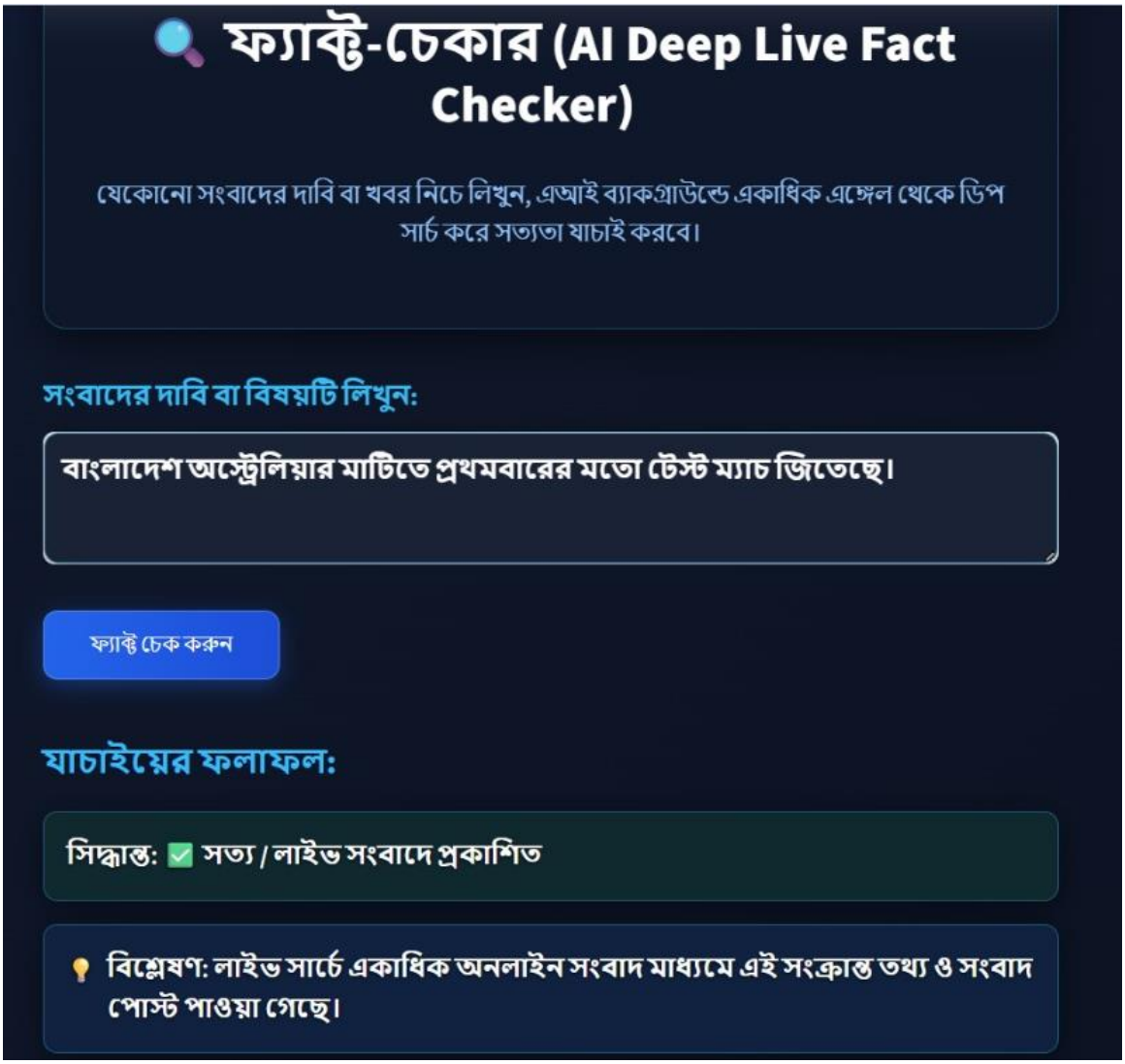


Fig. 5.2: Verdict - Real / Reported Example

Fig. 5.2 shows the result for the claim "বাংলাদেশ অস্ট্রেলিয়ার মাটিতে প্রথমবারের মতো টেস্ট ম্যাচ জিতেছে" (Bangladesh won a Test match on Australian soil for the first time), which the system verified as সত্য/Real against live news coverage of the result, demonstrating the green st.success path described in Section 5.3.3.

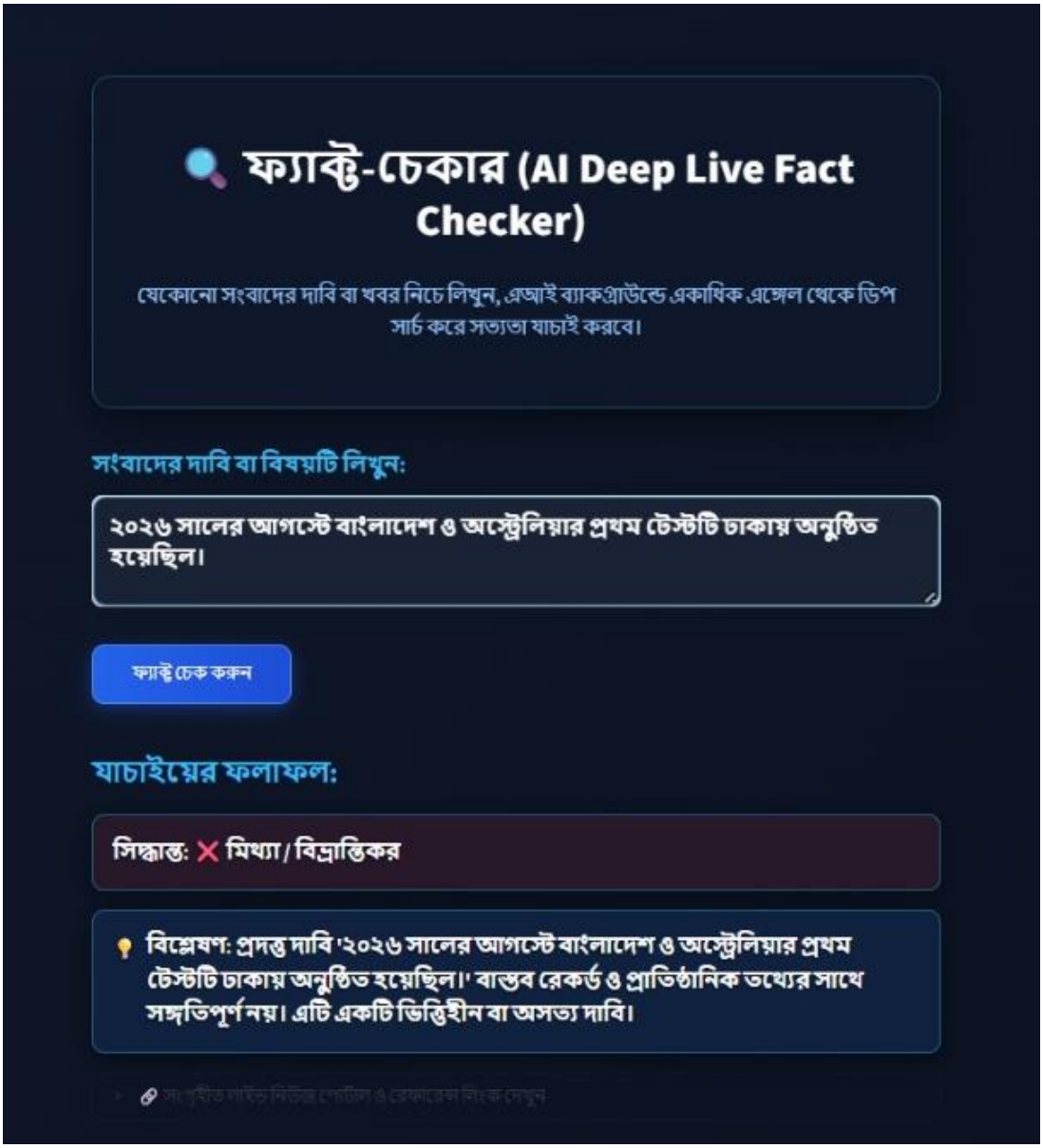


Fig. 5.3: Verdict - Fake / Misleading Example

Fig. 5.3 shows the result for a claim asserting that the same Bangladesh–Australia Test was played in Dhaka in August 2026 - a fabricated variation of the real event, constructed deliberately to test whether the system could distinguish a genuine result from a plausible-sounding but false variation of it. The system correctly flagged this as মিথ্যা/Fake, since the retrieved evidence placed the match in Darwin, not Dhaka.

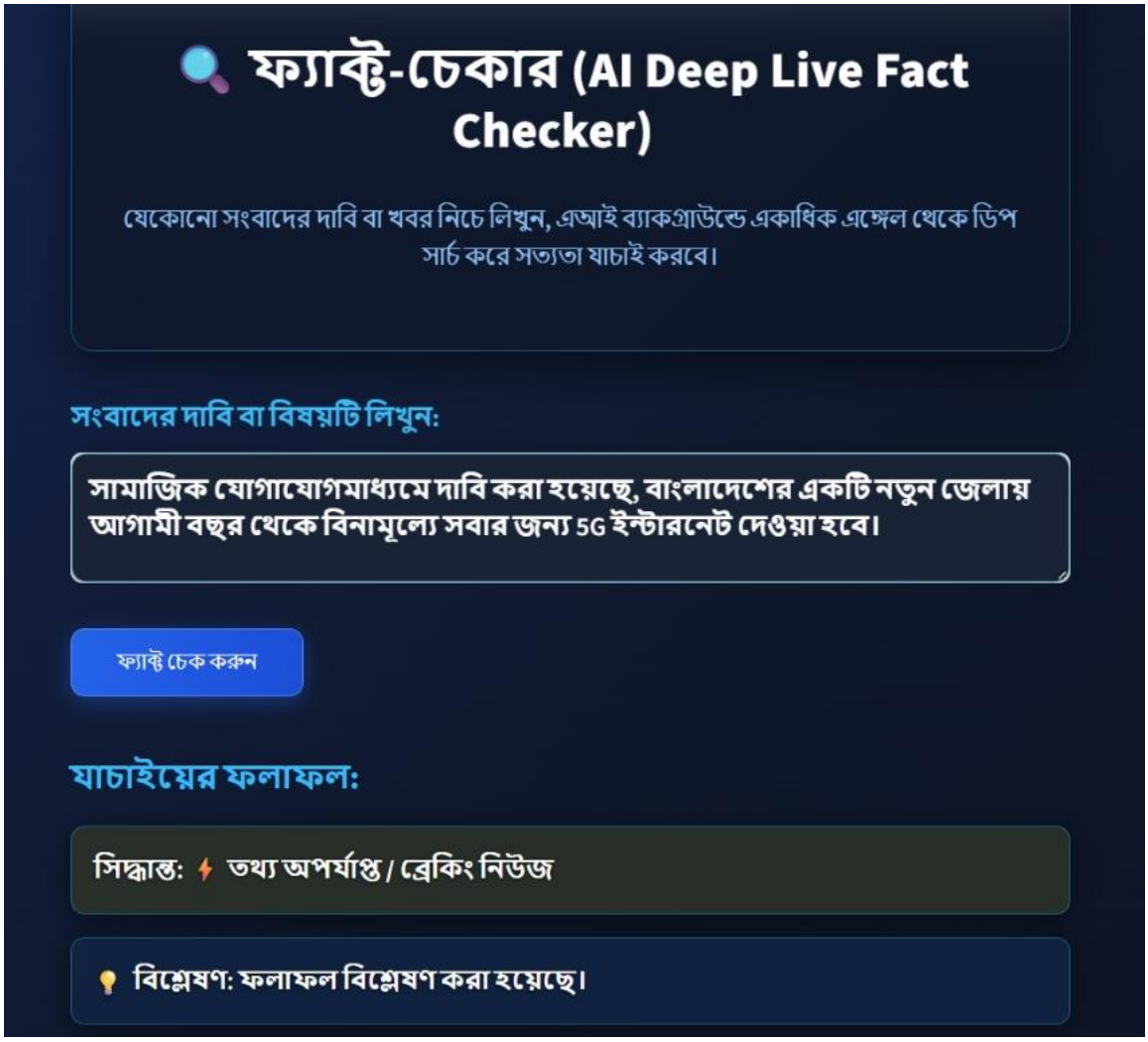


Fig. 5.4: Verdict - Developing / Unverified Example

Fig. 5.4 shows the result for a speculative claim about free 5G internet being rolled out in a new district of Bangladesh - a claim with no clear live news coverage either confirming or denying it, correctly returned as a Developing/Unverified (⚡) verdict rather than being forced into a Real or Fake label, which is the behaviour the four-rule verdict prompt in Section 5.2.4 is specifically designed to produce in the absence of clear evidence either way.

Fig. 5.2 through Fig. 5.4 illustrate the three possible verdict states described in Section 5.2.4, and correspond directly to the st.success / st.error / st.warning branching logic in Section 5.3.3, giving a visual confirmation that the parsing and colour-mapping logic implemented in the front end behaves as intended across all three cases rather than only in the simplest, most favourable one.

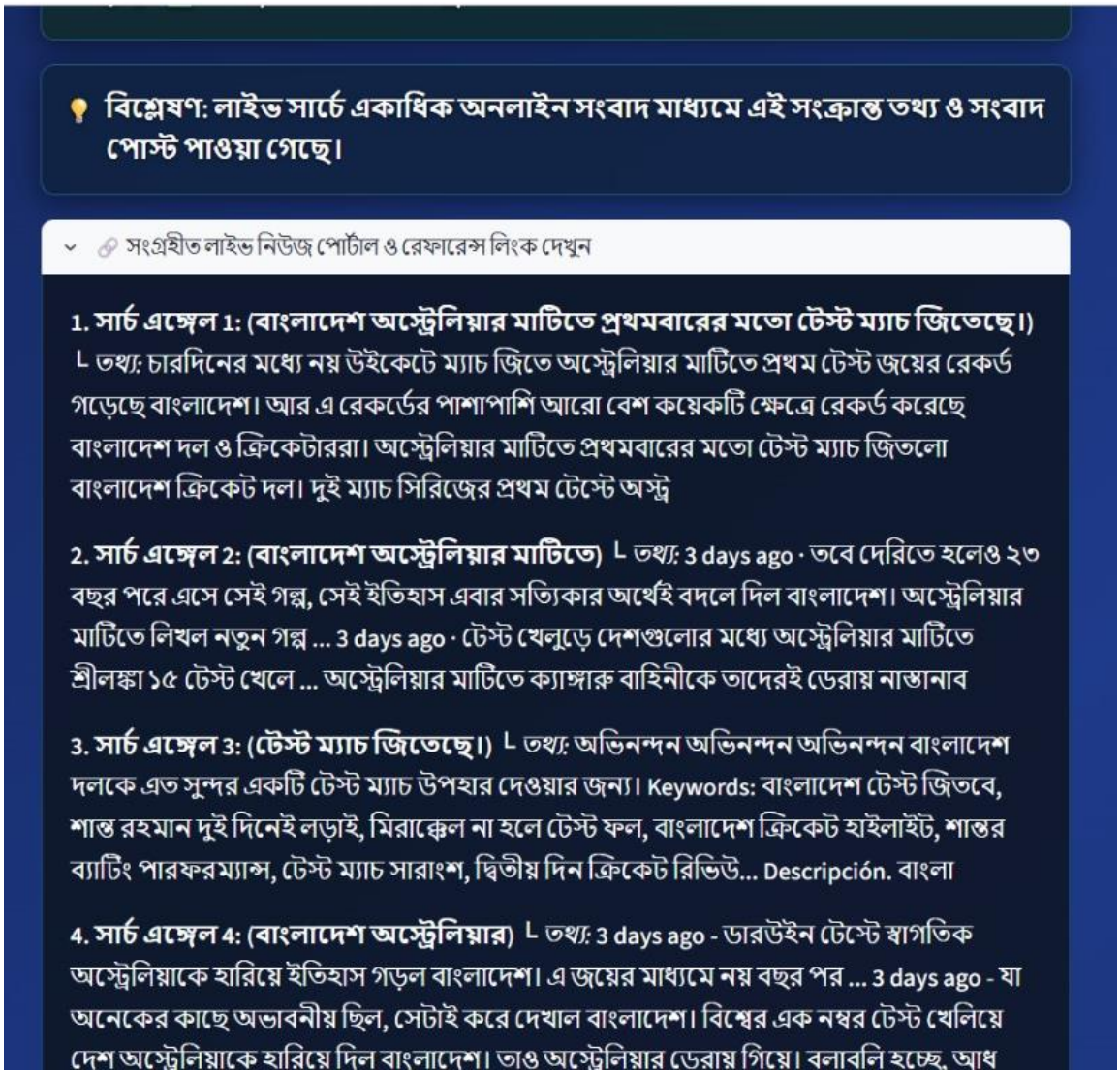


Fig. 5.5: Evidence Sources - Real Claim

Fig. 5.5 shows the evidence expander opened for the true claim in Fig. 5.2: the retrieved sources directly report Bangladesh's Test win in Australia, giving the verdict clear supporting evidence that a user could, if they wished, follow through to the original articles and verify independently.

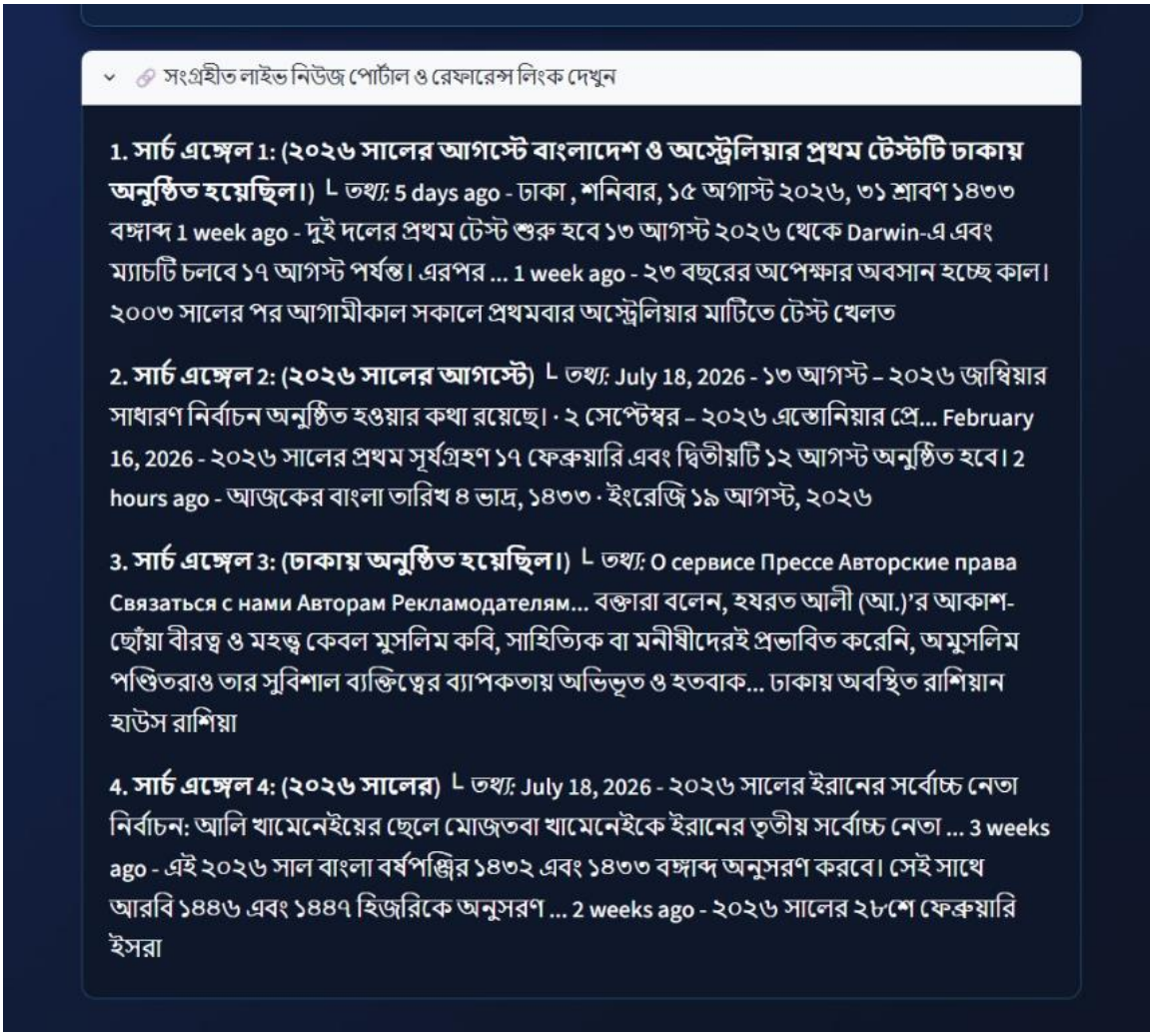


Fig. 5.6: Evidence Sources - Fake Claim

Fig. 5.6 shows the evidence expander for the fabricated Dhaka-Test claim in Fig. 5.3. Notably, the retrieved sources do not corroborate the claim at all - they describe the real Darwin fixture, unrelated election coverage, and other unconnected news. This mismatch between the claim and the retrieved evidence is itself part of what the LLM uses to reach a Fake verdict, and illustrates why Section 5.3.3 surfaces the evidence panel to the user rather than only showing the final label: the absence of corroborating evidence is, in this case, as informative as a direct contradiction would have been.

CHAPTER 6

LIMITATIONS AND FUTURE WORKS

6.1 Limitations

While LiveFactChecker meets its core objective of producing evidence-grounded verdicts for Bengali, English, and Banglish claims, the current implementation has a number of limitations, arising both from deliberate scope decisions made to keep the project achievable within a single academic term, and from constraints discovered during development and documented in the walkthrough in Chapter 5. None of these limitations undermine the core retrieval-augmented approach itself; rather, they describe specific engineering gaps between the current proof-of-concept and a system that could be relied on in a production setting with many concurrent users.

- **Hardcoded API Credentials:** GROQ_API_KEY and TAVILY_API_KEY are currently defined as string literals at the top of fact_checker.py rather than loaded from environment variables or Streamlit secrets. This is a security risk if the code is shared or committed to a public repository, since the keys would be exposed alongside the source.
- **Silent Exception Handling:** Several except Exception: pass blocks in generate_multi_angle_queries() and fetch_evidence() suppress errors entirely rather than logging them, which kept the user-facing experience smooth during development but makes it difficult to diagnose why a particular query or search call failed after the fact.
- **Sequential Retrieval Latency:** A single fact-check performs up to ten Tavily search calls, a possible DuckDuckGo fallback, and two separate Groq LLM calls, all executed sequentially rather than concurrently. This can make a single check take several seconds, which is acceptable for a thesis demonstration but would not scale well to concurrent multi-user traffic.
- **Fragile Verdict-to-UI Mapping:** app.py decides which coloured box to render by checking whether the substrings সত্য or মিথ্যা appear anywhere in the verdict string returned by the LLM, rather than matching against a fixed set of enum values. An

unexpected phrasing from the model - for example a partially-true verdict - could be mis-colored by this substring check.

- **No Quantitative Accuracy Evaluation:** The claims used in Chapter 5 (Fig. 5.1–5.6) demonstrate that the system behaves correctly on a small number of hand-picked examples, but the project does not include a systematic evaluation against a larger, labeled set of true and false Bengali claims, so no precision or recall figure can be reported.
- **Weaker Fallback Evidence Quality:** DuckDuckGoSearchRun returns a single block of summarized text rather than Tavily's structured title/content/url results, so evidence retrieved through the fallback path cannot be shown to the user as a clickable source link, only as an unattributed text snippet.
- **Third-Party API Dependency:** The system's availability and cost are entirely dependent on the uptime, rate limits, and pricing of Groq and Tavily; if either service becomes unavailable or changes its terms, the system stops functioning until the code is updated to point at an alternative provider.
- **No Caching of Repeated Claims:** Checking the same claim twice re-runs the entire pipeline - two LLM calls and up to ten search calls - rather than reusing a previous result, which unnecessarily consumes API quota for claims that are checked more than once.
- **Prompt Injection Exposure:** The raw claim text is interpolated directly into both the query-generation and verdict prompts without sanitisation, so an adversarially crafted claim could, in principle, attempt to override the prompt's instructions.

6.2 Future Works

Each limitation identified in Section 6.1 points to a concrete, scoped improvement rather than to a fundamental rethink of the system's design. These are listed below roughly in the order they would need to be addressed in a practical development roadmap - configuration and reliability fixes first, since they are prerequisites for any wider deployment, followed by evaluation and quality improvements that would strengthen confidence in the system's accuracy, followed finally by broader deployment once the earlier concerns have been resolved.

- **Externalised Configuration:** Move GROQ_API_KEY and TAVILY_API_KEY out of source code and into environment variables or Streamlit's st.secrets mechanism, so credentials are never committed alongside the code.
- **Structured Logging:** Replace the silent except Exception: pass blocks with proper logging of failures, so that query-generation or search failures can be diagnosed after the fact instead of failing invisibly.
- **Concurrent Evidence Retrieval:** Replace the sequential loop in fetch_evidence() with concurrent requests (for example, via concurrent.futures.ThreadPoolExecutor or an async HTTP client), which should substantially reduce end-to-end latency without changing the retrieval logic itself.
- **Structured LLM Output:** Replace the free-text VERDICT:/EXPLANATION: parsing convention with a structured output mode (JSON mode or function calling, where supported by the Groq API), removing the current reliance on string-prefix matching in both fact_checker.py and app.py.
- **Result Caching:** Cache verify_claim() results for identical or near-identical claims (for example, with st.cache_data or an external cache) so repeated checks of the same claim do not repeat the full API cost.
- **Benchmark Dataset and Evaluation:** Assemble a labeled set of true and false Bengali claims - potentially drawn from existing fact-checking organisations' archives - and report precision, recall, and verdict-agreement metrics rather than relying on illustrative examples alone.
- **Source Credibility Weighting:** Extend fetch_evidence() to weight or filter sources by publisher reliability, rather than treating every Tavily or DuckDuckGo result as equally trustworthy evidence.
- **Broader Deployment:** Package the system for public deployment (for example, via Streamlit Community Cloud) so it can be used beyond a local demonstration, once the credential-handling and rate-limiting concerns above are addressed.

CHAPTER 7

CONCLUSION

The project created LiveFactChecker, which is a fact-checking tool based on evidence for Bengali speakers that does not depend on the static knowledge possessed by a language model. In contrast, it fetches live web evidence when the claim is being checked. This is important because the fact-checking research is dominated by English and faces the problems of static knowledge and hallucinations associated with stand-alone LLMs. The project takes Bengali, English, and Banglish claims, creates multiple bilingual search queries, gets the evidence either confirming or contradicting the claim via Tavily using DuckDuckGo as a fallback, and gives a verdict with a Bengali explanation via Streamlit without any setup.

The results presented in Chapter 5 proved that the system correctly classified a true Bangladesh Test cricket claim, marked a fabricated variation of the claim, which said that the match took place in Dhaka, as fake, and gave the Developing verdict to a speculational claim without evidence. The evidence panels also confirmed that false claims can lack any evidence but not contradicting evidence.

This was able to meet its main aim of developing an effective, evidence-based, multi-lingual and multi-script fact-checking pipeline, and proved the feasibility of transferring retrieval-augmented models which have been developed mainly within English contexts into other languages without a drastically different architecture design. Nevertheless, Chapter 6 raised nine constraints relating to configuration security, error handling, latency, evaluation, and reliance on third party services.

Potential solutions include storing API credentials externally and conducting benchmark evaluation through labeling Bengali claims.

Overall, LiveFactChecker is a proof of concept in the development of evidence-based Bengali fact-checking, while at the same time being a means towards building a more secure and evaluated product.

REFERENCES

[1] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. Advances in Neural Information Processing Systems (NeurIPS), 33.

[2] Thorne, J., Vlachos, A., Christodoulopoulos, C., & Mittal, A. (2018). FEVER: a Large-scale Dataset for Fact Extraction and VERification. Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT).

[3] Hassan, N., Arslan, F., Li, C., & Tremayne, M. (2017). Toward Automated Fact-Checking: Detecting Check-worthy Factual Claims by ClaimBuster. Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.

[4] Guo, Z., Schlichtkrull, M., & Vlachos, A. (2022). A Survey on Automated Fact-Checking. Transactions of the Association for Computational Linguistics, 10, 178–206.

[5] Shu, K., Sliva, A., Wang, S., Tang, J., & Liu, H. (2017). Fake News Detection on Social Media: A Data Mining Perspective. ACM SIGKDD Explorations Newsletter, 19(1), 22–36.

[6] Zhou, X., & Zafarani, R. (2020). A Survey of Fake News: Fundamental Theories, Detection Methods, and Opportunities. ACM Computing Surveys, 53(5), 1–40.

[7] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention Is All You Need. Advances in Neural Information Processing Systems (NeurIPS), 30.

[8] Touvron, H., Martin, L., Stone, K., et al. (2023). Llama 2: Open Foundation and Fine-Tuned Chat Models. Meta AI Research.

[9] Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y. J., Madotto, A., & Fung, P. (2023). Survey of Hallucination in Natural Language Generation. ACM Computing Surveys, 55(12), 1–38.

[10] Streamlit Inc. (2024). Streamlit Documentation. https://docs.streamlit.io

[11] Groq Inc. (2024). Groq API Documentation. https://console.groq.com/docs

[12] Tavily AI. (2024). Tavily Search API Documentation. https://docs.tavily.com

[13] Popat, K., Mukherjee, S., Strötgen, J., & Weikum, G. (2017). Where the Truth Lies: Explaining the Credibility of Emerging Claims on the Web and Social Media. Proceedings of the 26th International Conference on World Wide Web Companion.

[14] Zellers, R., Holtzman, A., Rashkin, H., Bisk, Y., Farhadi, A., Roesner, F., & Choi, Y. (2019). Defending Against Neural Fake News. Advances in Neural Information Processing Systems (NeurIPS), 32.

[15] Nakov, P., Corney, D., Hasanain, M., Alam, F., Elsayed, T., Barrón-Cedeño, A., Papotti, P., Shaar, S., & Da San Martino, G. (2021). Automated Fact-Checking for Assisting Human Fact-Checkers. Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI), Survey Track.

[16] Bhattacharjee, A., Hasan, T., Ahmad, W. U., Mubasshir, K. S., Islam, M. S., Iqbal, A., Rahman, M. S., & Shahriyar, R. (2022). BanglaBERT: Language Model Pretraining and Benchmarks for Low-Resource Language Understanding Evaluation in Bangla. Findings of the Association for Computational Linguistics: NAACL 2022.

[17] Sitaram, S., Chandu, K. R., Rallabandi, S. K., & Black, A. W. (2019). A Survey of Code-switched Speech and Language Processing. arXiv preprint arXiv:1904.00784.

[18] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., et al. (2020). Language Models are Few-Shot Learners. Advances in Neural Information Processing Systems (NeurIPS), 33.

[19] OpenAI. (2023). GPT-4 Technical Report. arXiv preprint arXiv:2303.08774.

[20] Bang, Y., Cahyawijaya, S., Lee, N., Dai, W., Su, D., Wilie, B., Lovenia, H., Ji, Z., Yu, T., Chung, W., Do, Q. V., Xu, Y., & Fung, P. (2023). A Multitask, Multilingual, Multimodal Evaluation of ChatGPT on Reasoning, Hallucination, and Interactivity. Proceedings of the 13th International Joint Conference on Natural Language Processing (IJCNLP-AACL).